SAS® 9.4 FedSQL Language Reference, Fifth Edition

SAS® Documentation
March 2, 2023

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2016. *SAS® 9.4 FedSQL Language Reference, Fifth Edition*. Cary, NC: SAS Institute Inc.

SAS® 9.4 FedSQL Language Reference, Fifth Edition

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

March 2023

9.4-P11:fedsqlref

Contents

What's New in SAS 9.4 FedSQL Language Reference

Overview

FedSQL is a SAS proprietary implementation of the ANSI SQL:1999 core standard. It is a vendor-neutral SQL dialect that provides support for ANSI SQL data types and other ANSI 1999 core compliance features and proprietary extensions. FedSQL can access data from various data sources without having to submit queries in the SQL dialect that is specific to the data source. In addition, a single FedSQL query can target data in several data sources and return a single result set. FedSQL is included in SAS 9.4 and SAS Viya.

The FEDSQL procedure enables you to submit FedSQL language statements from a Base SAS session. The fedSql.execDirect action enables you to submit FedSQL language statements from CAS. For information about what's new in FedSQL in CAS, see *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*.

SAS 9.4M8

The documentation is updated to list aggregate functions that are supported with Salesforce and to describe how they are processed. The behavior is dependent on the setting of the BULK_API= connection option, which is new in SAS/ACCESS software. For more information, see "Overview of Aggregate Functions" on page 218. For information about the BULK_API= option, see *SAS/ACCESS for Nonrelational Databases: Reference*.

July 2021

FedSQL statement support for MongoDB is enhanced in the latest version of SAS/ACCESS Interface to MongoDB software. You can now perform the following tasks when manipulating and querying MongoDB collections with FedSQL:

- specify subqueries as inner queries in the DELETE statement
- use query expressions in the INSERT statement
- update date and time values with the UPDATE statement
- specify subqueries as inner queries in the UPDATE statement

Note: The UPDATE statement still does not allow the use of subqueries as inner queries to update child tables.

April 2021

The mapping of the TIMESTAMP(*p*) data type for Google BigQuery has changed in the latest SAS/ACCESS Interface to Google BigQuery software update. For more information, see "Data Types for Google BigQuery" on page 1120.

SAS 9.4M7

SAS 9.4M7 has the following changes and enhancements:

- The FedSQL language adds support for Spark and Yellowbrick as data sources when appropriate SAS/ACCESS software is installed. For information about FedSQL statement support for the databases, see "FedSQL Statements" on page 903. For data type support, see Appendix 2, "Data Type Reference," on page 1109.
- A new table option, TD_UNICODE_PASSTHRU=, is available for use with Teradata. A bulk-loading option, TD_UNICODE_PASSTHRU= allows pass-through characters (PTCs) to be imported into and exported from Teradata. For more information, see "TD_UNICODE_PASSTHRU= Table Option" on page 1093.
- The TD_DATA_ENCRYPTION= table option supports aliases. For more information, see "TD_DATA_ENCRYPTION= Table Option" on page 1072.

SAS Viya 3.5

SAS Viya 3.5 has the following changes and enhancements:

- The FedSQL language adds support for the MongoDB and Salesforce nonrelational databases when appropriate SAS/ACCESS software is installed. The functionality is available when accessing the databases through a SAS library. It is also available when accessing the databases through the CAS server. For information about FedSQL statement and data type support when accessing the databases through a SAS library, see "FedSQL Statements" on page 903 and Appendix 2, "Data Type Reference," on page 1109. For statement and data type support on the CAS server, see *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*.
- There are improvements to data type support for various third-party relational databases when appropriate SAS/ACCESS software is installed. These features are available when the databases are accessed through a SAS library.
  - FedSQL now creates VARCHAR columns containing greater than 65,535 characters as type STRING in Hive. For more information, see "Data Types for Hive" on page 1125.
  - FedSQL now reads and writes Teradata NUMBER columns. For more information, see "Data Types for Teradata" on page 1159.
  - FedSQL now reads JSON columns in MySQL. For more information, see "Data Types for MySQL" on page 1137.
  - You no longer have to set the TIMEZONE system option to display date and time data as it was recorded in Salesforce.
- The COMPRESS function supports a Modifier argument. See "COMPRESS Function" on page 375.
- The documentation was enhanced:
  - The documentation describing the return values of FedSQL functions that are like-named in Base SAS has been updated. See "Overview of FedSQL Functions" on page 214.
  - A new example shows when to use the FORCE option in the DROP TABLE statement. See "DROP TABLE Statement" on page 942.
  - The documented syntax for the SELECT statement UNION, EXCEPT, and INTERSECT operators has been modified. See the "SELECT Statement" on page 952. Usage information has also been added. See "Query Expressions and Subqueries" on page 46.
  - The documentation has been modified to remove mention of the FCMP procedure. The FCMP procedure is not supported for FedSQL.
  - A new appendix has been added. See Appendix 1, "Troubleshooting: Resolving FedSQL Errors," on page 1105.
  - Appendix 6, "How to Perform Common PROC SQL Tasks with FedSQL" has been updated to include three new topics: "Overwrite an Existing Table with

FedSQL" on page 1202, "Using Explicit Pass-Through" on page 1215, "Comparing Implicit SQL Pass-Through" on page 1216.

November 2019

Write support for MongoDB and Salesforce databases is available on SAS 9.4M6 when appropriate SAS/ACCESS software is installed. For information about FedSQL statement and data type support for the databases, see "FedSQL Statements" on page 903 and Appendix 2, "Data Type Reference," on page 1109. For information about how data from the nonrelational databases is written, see *SAS/ACCESS for Nonrelational Databases: Reference*

August 2019

The FedSQL language adds support for Google BigQuery and Snowflake as data sources when appropriate SAS/ACCESS software is installed. The data sources are supported both on SAS 9.4M6 and SAS Viya 3.4. Access is Read and Write, and in SAS Viya, through a SAS library or a caslib. The FedSQL language supports different functionality on the CAS server than it does in a SAS library. *SAS FedSQL Language Reference* describes the FedSQL statements, table options, and data types support that are available for the data sources when they are accessed through a SAS library. For information about the FedSQL statements, table options, and data types that are supported on the CAS server, see *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*.

May 2019

This documentation is enhanced with a new appendix: Appendix 6, "How to Perform Common PROC SQL Tasks with FedSQL," on page 1201.

April 2019

The FedSQL language adds support on SAS 9.4M6 for the MongoDB and Salesforce nonrelational databases as data sources when appropriate SAS/ACCESS software is installed. Access to the databases is Read-only. For

information about FedSQL statement and data type support for the databases, see "FedSQL Statements" on page 903 and Appendix 2, "Data Type Reference," on page 1109. For information about how data from the nonrelational databases is retrieved, see *SAS/ACCESS for Nonrelational Databases: Reference*

SAS 9.4M6

SAS 9.4M6 has the following changes and enhancements:

- The FedSQL language adds support as a data source for databases that are compliant with JDBC when appropriate SAS/ACCESS software is installed. Access is Read and Write. For information about FedSQL statement support, see "FedSQL Statements" on page 903. For information about data type support, see Appendix 2, "Data Type Reference," on page 1109.
- The FedSQL language creates VARCHAR columns containing greater than 65,535 characters as type STRING in Hive. Previously, the columns were created as VARCHAR, regardless of length. See "Data Types for Hive" on page 1125.

The documentation was modified as follows:

- The documentation for the CREATE TABLE statement was modified to include naming restrictions for HDMD and Hive data sources. For more information, see "CREATE TABLE Statement" on page 915.
- The documentation for the DESCRIBE VIEW statement was modified to clarify support for SAS data views. See "DESCRIBE VIEW Statement" on page 940.
- The documentation clarifies support for SAS system options. See Chapter 9, "FedSQL and System Options," on page 1101.

SAS Viya 3.4

SAS Viya 3.4 has the following changes and enhancements:

- The FedSQL language adds support as data sources databases that are compliant with JDBC when appropriate SAS/ACCESS software is installed. These databases are accessible by using a SAS library or on the CAS server. The FedSQL language supports different functionality on the CAS server than it does in a SAS library. *SAS FedSQL Language Reference* describes the statements, table options, and data type support that is available when accessing the data sources through a SAS library. For information about the FedSQL statements, table options, and data types that are available on the CAS server, see *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*.
- The FedSQL language supports explicit SQL pass-through on the CAS server. For more information, see *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*. For an example of how an explicit pass-through request is

submitted in the fedSql.execDirect action, see *SAS Viya: System Programming Guide*.

- The SYSGET function returns the value of the specified operating environment variable.
- The URLDECODE and URLENCODE functions are documented for FedSQL.

SAS Viya 3.3

The FedSQL language supports SAS library access to all SAS 9.4 third-party data sources in SAS Viya when appropriate SAS/ACCESS software is installed. It supports CAS library access to all data sources for which SAS Data Connector software is installed. The FedSQL language supports different functionality on the CAS server than it does in a SAS library. *SAS FedSQL Language Reference* describes the statements, table options, and data type support that is available when accessing the data sources through a SAS library. For information about the FedSQL statements, table options, and data types that are available on the CAS server, see *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*.

SAS 9.4M5

SAS 9.4M5 has the following changes and enhancements:

- If you have SAS Viya configured in addition to SAS 9.4, you can submit FedSQL statements to SAS Cloud Analytic Services from your SAS 9.4 session. For more information, see "Running FedSQL Programs" on page 5 and "FedSQL and SAS Cloud Analytic Services" on page 7.
- The FedSQL language adds support for Amazon Redshift, Microsoft SQL Server, and Vertica data sources on SAS 9.4M5 when appropriate SAS/ACCESS software is installed. For information about FedSQL statement support for the databases, see "FedSQL Statements" on page 903. For information about table options, see "FedSQL Statement Table Options by Data Source" on page 991. For information about data type support for each data source, see Appendix 2, "Data Type Reference," on page 1109.
- You can now get information about views from DICTIONARY.COLUMNS queries for a SAS library. See "DICTIONARY Tables" on page 66.
- The FedSQL language adds support for FINANCE functions. For more information, see "FINANCE Function" on page 443.
- The ENCRYPT= table option supports stronger AES encryption for SAS data sets. For more information, see "ENCRYPT= Table Option" on page 1039.
- SAS Scalable Performance Data (SPD) Server tables can now be encrypted with ENCRYPT= and ENCRYPTKEY= table options.

- PRE_TABLE_OPTS= and POST_TABLE_OPTS= table options for Hive. See "PRE_TABLE_OPTS= Table Option" on page 1065 and "POST_TABLE_OPTS= Table Option" on page 1064.
- Support for the following functions:

ANYALNUM	CSC	INTTEST	NWKDOM
ANYALPHA	CUMIPMT	INTZ	PDF
ANYDIGIT	COMPRINC	IPMT	PERM
ANYFIRST	DAIRY	JBESSEL	PMT
ANYLOWER	DEQUOTE	LCOMB	PROBMC
ANYNAME	DEVIANCE	LEFT	PROBMED
ANYPUNCT	DUR	LFACT	PVP
ANYSPACE	DURP	LOG1PX	QUANTILE
ANYUPPER	FACT	LOGCDF	ROUNDZ
ANYXDIGIT	FMTINFO	LOGISTIC	SAVING
AIRY	FNONCT	LOGPDF	SDF
ARCOSH	GARKHCLPRC	LOGSDF	SEC
ARSINH	GARKHPTPRC	MOD	SQUANTILE
ARTANH	HOLIDAY	MODZ	SUMABS
BNOT	IBESSEL	NOMRATE	TIMEVALUE
CDF	INPUTC	NOTALNUM	TNONCT
CNONCT	INPUTN	NOTALPHA	TRANSTRN
COMB	INT	NOTDIGIT	TRANWRD
COMPARE	INTCINDEX	NOTFIRST	TRIGAMMA
COMPBL	INTCYCLE	NOTLOWER	TRIM
CONSTANT	INTFIT	NOTNAME	WHICHC
CONVX	INTGET	NOTPUNCT	WHICHN
CONVXP	INTINDEX	NOTSPACE	YIELDP
COUNTC	INTSEAS	NOTUPPER	YRDIF
COUNTW	INTSHIFT	NOTXDIGIT	

For more information, see "FedSQL Functions" on page 207.

SAS 9.4M3

The following functionality was added to the FedSQL language:

- The FedSQL language adds support for HAWQ and Impala distributions of Hadoop when appropriate SAS/ACCESS software is installed. For information about FedSQL statement support for the databases, see "FedSQL Statements" on page 903. For information about table options, see "FedSQL Statement Table Options by Data Source" on page 991. For information about data type support for each data source, see Appendix 2, "Data Type Reference," on page 1109.

- DICTIONARY.TABLES queries have been enhanced to return SECURITY and SERVICE columns. For a complete list of the columns returned by DICTIONARY.TABLES queries, see "DICTIONARY.TABLES" on page 1198.
- "DBCREATE_TABLE_OPTS= Table Option" on page 1035 for Hive.
- "ENCODING= Table Option" on page 1038 for SAS data sets.
- Expanded data type support for the SAS HDMD data source, including support for FLOAT and for reading and writing the DECIMAL/NUMERIC(p,s) data types. See "Data Types for HDMD" on page 1124 for a complete list of supported data types.
- Expanded data type support for Hive:
  - support for reading the Hive ARRAY, MAP, STRUCT, and UNION complex types.
  - support for reading and writing the DECIMAL/NUMERIC(p,s) and BINARY data types. Full support of the DECIMAL/NUMERIC and BINARY types is provided in Hive 0.13 and later. In earlier Hive versions, DECIMAL/NUMERIC columns are not supported, and BINARY columns can be created but not retrieved.
  - support for writing the VARBINARY type. Columns of type VARBINARY type are created as type BINARY in Hive.
  - support for reading and writing TIME values.

  For a complete list of the data types that are support for Hive, see "Data Types for Hive" in Appendix 2, "Data Type Reference," on page 1109.
- Hive has been enhanced to search tables that have more than one column of type TIME with the HOUR, MINUTE, and SECOND functions. See "HOUR Function" on page 549, "MINUTE Function" on page 649, and "SECOND Function" on page 806.
- MDS now supports UNIQUE and UNIQUE(COLUMN,[COLUMN]) syntax in the CREATE TABLE statement. See "CREATE TABLE Statement" on page 915.
- Documentation enhancements:
  - clarification of DICTIONARY support for third-party DBMS. See "DICTIONARY Tables" on page 66.
  - clarification of Hive and SAS data set support of the CHAR(n) [CHARACTER SET "character-set-identifier"] syntax in the CREATE TABLE statement. See "CREATE TABLE Statement" on page 915.
  - clarification of SAS data set and SPD Engine data set support of the CREATE INDEX ASCENDING and DESCENDING arguments. See "CREATE INDEX Statement" on page 913.
  - clarification of support for the TIME data type in ODBC-style interfaces. See "Data Types for Greenplum" on page 1121, "Data Types for HAWQ" on page 1122, and "Data Types for Netezza" on page 1139.
  - clarification of multi-byte support in the CHAR(n) and VARCHAR(n) data types for ODBC data sources. See "Data Types for ODBC" on page 1140.
  - Documented the minimum and maximum year values supported for the DATE type in Hive. See "Data Types for Hive" on page 1125.

- Corrected information about support for FORMAT=, INFORMAT=, and LABEL= modifiers in Appendix 7, “FedSQL and the ANSI Standard,” on page 1227. These modifiers are not supported in the SELECT statement.

SAS 9.4M2

The following functionality was added to the FedSQL language:

- The FedSQL language adds support for Hive, SAS HDMD, and PostgreSQL data sources when appropriate SAS/ACCESS software is installed. For information about FedSQL statement support for the databases, see “FedSQL Statements” on page 903. For information about table options, see “FedSQL Statement Table Options by Data Source” on page 991. For information about data type support for each data source, see Appendix 2, “Data Type Reference,” on page 1109.
- “CAST Function” on page 310
- “DBCREATE_INDEX_OPTS= Table Option” on page 1034 for ODBC

SAS 9.4M1

The following functionality was added to the FedSQL language:

- The FedSQL language adds support for Memory Data Store (MDS) and SAP HANA data sources when appropriate SAS/ACCESS software is installed. For information about FedSQL statement support for the databases, see “FedSQL Statements” on page 903. For information about table options, see “FedSQL Statement Table Options by Data Source” on page 991. For information about data type support for each data source, see Appendix 2, “Data Type Reference,” on page 1109.
- Rename table and rename column functionality in the ALTER TABLE statement. See “ALTER TABLE Statement” on page 905.

The documentation was enhanced to include the following:

- examples of join operations. See “Join Operations” on page 30.
- examples of EXCEPT, INTERSECT, and UNION operations. See “Default Query Expression Behavior” on page 47.
- examples of subqueries. See “Overview of Subqueries” on page 51, “Examples of Correlated Subqueries” on page 51, “Examples of Scalar Subqueries” on page 52, and “Examples of Non-Correlated Queries” on page 54.
- an explanation of how the PUT function can be used. See “Using Formats in FedSQL” on page 78 and “Using a User-Defined Format with the PUT Function” on page 80.

- an explanation of how FedSQL handles transactions. See "Transactions in FedSQL" on page 70.

PART 1

Introduction

1

Introduction to the FedSQL Language

Introduction to the FedSQL Language

SAS FedSQL is a SAS proprietary implementation of ANSI SQL:1999 core standard. It provides support for new data types and other ANSI 1999 core compliance features and proprietary extensions. FedSQL provides a scalable, threaded, high-performance way to access, manage, and share relational data in multiple data sources. When possible, FedSQL queries are optimized with multi-threaded algorithms in order to resolve large-scale operations. Beginning in April 2019, FedSQL also supports reading of some non-relational data sources.

For applications, FedSQL provides a common SQL syntax across all data sources. That is, FedSQL is a vendor-neutral SQL dialect that accesses data from various data sources without having to submit queries in the SQL dialect that is specific to

the data source. In addition, a single FedSQL query can target data in several data sources and return a single result set.

For more information about SQL:1999 core-compliant syntax and SAS extensions, see Appendix 7, “FedSQL and the ANSI Standard,” on page 1227.

Benefits of FedSQL

FedSQL provides many benefits if you are working in an environment in which you need more features than are provided in the SQL procedure.

- FedSQL conforms to the ANSI SQL:1999 core standard. This conformance allows it to process queries in its own language and the native languages of other data sources that conform to the standard.
- FedSQL supports many more data types than previous SAS SQL implementations. Traditional data source access through SAS/ACCESS translates target data source data types to and from two legacy SAS data types, which are SAS numeric and SAS character. When FedSQL connects to a data source, the language matches or translates the target data source’s definition to these data types, as appropriate, which allows greater precision. Supported data types are described in “Data Types” on page 14.
- FedSQL handles federated queries. With the traditional DATA step or the SQL procedure, a SAS/ACCESS LIBNAME engine can access only the data of its intended data source.
- The FedSQL language can create data in any of the supported data sources, even if the target data source is not represented in a query. This enables you to store data in the data source that most closely meets the needs of your application.

Federated Queries

A federated query is one that accesses data from multiple data sources and returns a single result set. The data remains stored in the data source. For example, in this query, data is requested from an Oracle table and from two Teradata tables:

```
select Ora1.city  Ora1.State, Ora1.zip
from Oracle.Tbl1 Ora1, Teradata.Tbl2 Tera2, Teradata.Tbl3 Tera3
where Ora1.zip = Tera2.zip and Tera2.zip = Tera3.zip;
```

Running FedSQL Programs

You can submit FedSQL programs in one of the following ways:

- Through the SAS windowing environment or SAS Studio by using the FEDSQL procedure. The FedSQL procedure can be used to submit FedSQL statements in SAS 9.4, SAS Viya, and on the SAS Cloud Analytic Services (CAS) server. See *Base SAS Procedures Guide*.
- Through the FedSQL.execDirect action to the CAS server. The FedSQL execDirect action can be called from a SAS Studio session or from a Python, Lua, or R program. In SAS Studio, the FedSQL.execDirect action is used with the CAS procedure. For more information about the FedSQL.execDirect action, see *SAS Viya: System Programming Guide*.
- From a JDBC, ODBC, or OLE DB client by using SAS Federation Server. See *SAS Federation Server: Administrator's Guide*.
- Through the SAS windowing environment or SAS Studio to the SAS Federation Server using the SAS LIBNAME engine for SAS Federation Server. You submit FedSQL statements to this LIBNAME engine in the PROC SQL EXECUTE statement. See *SAS LIBNAME Engine for SAS Federation Server: User's Guide*.
- From a DS2 program. See Appendix 3, "Using FedSQL and DS2," on page 1167.

Data Source Support

FedSQL can access the following data sources:

- Amazon Redshift
- Aster
- DB2 for UNIX and Windows operating environments
- CAS libraries
- Google BigQuery on Linux for x64
- Greenplum
- HAWQ
- Hadoop (Hive and HDMD)
- Impala
- databases that are compliant with JDBC
- Memory Data Store (MDS)
- Microsoft SQL Server
- MongoDB on Linux for x64

- MySQL
- Netezza
- databases that are compliant with ODBC
- Oracle
- PostgreSQL
- Salesforce
- SAP (Read-only)
- SAP HANA
- SAP IQ
- SAS data sets
- SAS Scalable Performance Data Engine (SPD Engine) data sets
- SAS Scalable Performance Data Server (SPD Server) tables in UNIX and Windows operating environments
- Snowflake on Linux for x64
- Spark
- Teradata for UNIX and Windows operating environments
- Vertica
- Yellowbrick

Note: Different data sources are supported in a Base SAS session, in SAS Cloud Analytic Services, and in SAS Federation Server. See "FEDSQL Procedure" in *Base SAS Procedures Guide* for data source support in Base SAS and in CAS. For information about the data sources that are supported in SAS Federation Server, see *SAS Federation Server: Administrator's Guide*.

Data Source Connection

In order to submit FedSQL statements to a data source, you must establish a connection to a data source.

The FEDSQL procedure generates a connection string by using the attributes of currently assigned librefs by default. You first submit a LIBNAME statement for the data source that you want to access (for example, a Base SAS LIBNAME statement or a Hadoop LIBNAME statement) and then run PROC FEDSQL. For more information, see "FEDSQL Procedure" in *Base SAS Procedures Guide*.

The SAS Federation Server LIBNAME engine obtains a data source connection by connecting to a SAS Federation Server and specifying a DSN that is defined on SAS Federation Server. You specify server connection properties and the DSN in a LIBNAME statement. You must have been given the name of a DSN by a SAS Federation Server administrator, and be familiar with the names of the catalogs and schemas accessed through the DSNs in order to access data. For more information, see *SAS LIBNAME Engine for SAS Federation Server: User's Guide*

SAS Federation Server encapsulates the information that is needed to connect to a data source in a Data Source Name (DSN) definition. A DSN is metadata that contains connection details. DSN definitions are created by a SAS Federation Server administrator. In order to use a DSN, you must first connect to SAS Federation Server. For more information about connecting to SAS Federation Server and submitting DSNs, see *SAS Federation Server: Administrator's Guide*.

FedSQL and SAS Cloud Analytic Services

SAS Cloud Analytic Services is an in-memory processing environment. FedSQL functionality in CAS is limited. For information about the FedSQL statements, expressions, predicates, and table options that are supported in CAS, see *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*. FedSQL supports the same functions and formats in CAS, SAS 9.4, and SAS Viya.

Purpose of This Documentation

The *SAS FedSQL Language Reference* provides the following information for the Federated Query Language (FedSQL).

- Conceptual information about the FedSQL language
- Detailed reference information for the major language elements:
  - formats
  - functions
  - expressions, operators, and predicates
  - statements
  - table options

Intended Audience

This document is intended for the following users:

- Application developers who write client applications and need an understanding of the FedSQL language. They write applications that create tables, bulk load tables, manipulate tables, and query data.

- Database administrators who design and implement the client/server environment. They administer the data by designing the databases and setting up the data source metadata. That is, database administrators build the data model.
- SAS programmers who write and submit Base SAS code such as SAS procedures and the SAS DATA step language and now want to take advantage of the FedSQL language.
- Programmers who want to create and query persisted SAS and DBMS data from a SAS library.

Typographical Conventions

Type styles have special meanings in the documentation of the FedSQL syntax. The following list explains the style conventions for the syntax:

UPPERCASE BOLD
identifies SAS keywords such as the names of statements and functions (for example, CREATE TABLE).

UPPERCASE ROMAN
identifies arguments and values that are literals (for example, FROM).

italic
identifies arguments or values that you supply. Items in italics can represent user-supplied values that are either one of the following.

- nonliteral values that are assigned to an argument (for example, END=*variable*).
- nonliteral arguments (for example, AS *alias*).

If more than one of an item in italics can be used, the items are expressed as *item* [, …*item*].

`monospace`
identifies examples of SAS code.

Syntax Conventions

The *SAS FedSQL Language Reference* uses the Backus-Naur Form (BNF). Specifically, it uses the same syntax notation that is used by Jim Melton in *SQL:1999 Understanding Relational Language Components*.

The main difference between traditional SAS syntax and the syntax that is used in the SAS FedSQL language reference documentation is in how optional syntax arguments are displayed. In the traditional SAS syntax, angle brackets (< >) are used to denote optional syntax. In the FedSQL syntax, square brackets ([ ]) are used to denote optional syntax and angle brackets are used to denote non-terminal components.

The following symbols are used in the FedSQL syntax.

::=
: This symbol can be interpreted as “consists of” or “is defined as”.

< >
: Angle brackets identify a non-terminal component; that is, a syntax component that can be further resolved into lower level syntax grammar.

[ ]
: Square brackets identify optional arguments. Any argument that is not enclosed in square brackets is a required argument. Do not enter the square brackets unless they are preceded by a backward slash (\), which denotes that they are literal.

{ }
: Braces provide a method to distinguish required multi-word arguments. Do not enter the braces unless they are preceded by a backward slash (\), which denotes that they are literal.

|
: A vertical bar indicates that you can choose one value from a group. Values separated by bars are mutually exclusive.

…
: An ellipsis indicates that the argument or group of arguments that follow the ellipsis can be repeated any number of times. If the ellipsis and the following arguments are enclosed in square brackets, they are optional.

\
: A backward slash indicates that next character is a literal.

The following examples illustrate the syntax conventions that are described in this section. These examples contain selected syntax elements, not the complete syntax.

CREATE TABLE {*table* | _NULL_}
[\{OPTIONS *SAS-table-option=value* […*SAS-table-option=value*] \}]]
(<column-definition> […<column-definition> | <table-constraint>])
| AS *query-expression*
;
<column-definition>::=
column data-type [<column-constraint> <DEFAULT value> [HAVING [FORMAT format] [INFORMAT informat] [LABEL=’label’]]]

- **CREATE TABLE** is in uppercase bold because it is the name of the statement.
- *table* is italics because it is an argument that you can supply.
- _NULL_ is uppercase roman because it is a literal argument.
- The braces that are preceded by a backward slash (before and after the literal argument OPTIONS) indicate that the braces are literal. That is, the OPTIONS statement must be specified within braces.
- <column-definition> is in angle brackets because it is a non-terminal argument that is further resolved into lower level syntax grammar.
- The square brackets and ellipsis around the second instance of <column-definition> indicates that it is optional and you can repeat this argument any number of times, separated by commas.
- You can specify a <column-definition> or an AS *query-expression*, but not both.

- AS is in uppercase roman because it is a literal argument.
- <column-definition>::= illustrates a non-literal expansion.

PART 2

FedSQL Language Reference

2

FedSQL Language Concepts

Data Types

A data type is an attribute of every column in a table that specifies the type of data the column stores. For example, the data type is the characteristic of a piece of data that says it is a character string, an integer, a floating-point number, or a date or time. The data type also determines how much memory to allocate for the column's value.

The following table lists the data types that are supported by FedSQL. Note that not all data types are available for table storage on each data source.

Table 2.1 *FedSQL Data Types*

Data Type	Description
BIGINT	stores a large signed, exact whole number, with a precision of 19 digits. The range of integers is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Integer data types do not store decimal values; fractional portions are discarded.
BINARY(*n*)	stores fixed-length binary data, where *n* is the maximum number of bytes to store. The maximum number of bytes is required to store each value regardless of the actual size of the value.
CHAR(*n*)[1]	stores a fixed-length character string, where *n* is the maximum number of characters to store. Each character uses 1 byte of storage. The maximum number of characters is required to store each value regardless of the actual size of the value. If `char(10)` is

Data Type	Description
	specified and the character string is only five characters long, the value is right-padded with spaces.
DATE	stores a calendar date. A date literal is specified in the format *yyyy-mm-dd*: a four-digit year (0001 to 9999), a two-digit month (01 to 12), and a two-digit day (01 to 31). For example, the date September 24, 1975 is specified as `1975-09-24`. FedSQL complies with ANSI SQL:1999 standards regarding dates. However, not all data sources support the full range of dates. For example, dates between 0001-01-01 and 1582-12-31 are not valid dates for a SAS data set, an SPD Engine data set, and an SPD Server table.
DOUBLE	stores a signed, approximate, double-precision, floating-point number. Allows numbers of large magnitude and permits computations that require many digits of precision to the right of the decimal point.
DECIMAL| NUMERIC(*p*,*s*)	stores a signed, exact, fixed-point decimal number, with user-specified precision and scale. The precision and scale determine the position of the decimal point. The precision is the maximum number of digits that can be stored to the left and right of the decimal point, with a range of 1 to 52. The scale is the maximum number of digits that can be stored following the decimal point. The scale must be less than or equal to the precision. For example, `decimal(9,2)` stores decimal numbers up to nine digits, with a two-digit, fixed-point fractional portion, such as 1234567.89. **Note:** The DECIMAL data type is supported for defining a column, inserting data into the column, and fetch operations, unless the operations cannot be fully passed down to the data source. Operations that cannot be passed to the data source are described in "FedSQL Implicit Pass-Through Facility" on page 67. Under those conditions, a DECIMAL column is handled as DOUBLE, which can affect precision. A request either fully supports DECIMAL, or is converted to DOUBLE. A request must be capable of being fully passed down to the data source in order to use DECIMAL.
FLOAT(*p*)	stores a signed, approximate, single-precision,or double-precision, floating-point number. The user-specified precision determines whether the data type stores a single-precision or double-precision number. If the specified precision is equal to or greater than 25, the value is stored as a double-precision number, which is a DOUBLE. If the specified precision is less than 25, the value is stored as a single-precision number, which is a REAL. For example, `float(10)` specifies to store up to 10 digits, which results in a REAL data type.
INTEGER	stores a regular size signed, exact whole number, with a precision of 10 digits. The range of integers is -2,147,483,648 to 2,147,483,647. Integer data types do not store decimal values; fractional portions are discarded.

Data Type	Description
	Note: Integer division by zero does not produce the same result on all operating systems. It is recommended that you avoid integer division by zero.
NCHAR(*n*)[1]	stores a fixed-length character string such as CHAR but uses a Unicode national character set, where *n* is the maximum number of multibyte characters to store. Depending on the platform, Unicode characters use either two or four bytes per character and support all international characters.
NVARCHAR(*n*)[1]	stores a varying-length character string such as VARCHAR but uses a Unicode national character set, where *n* is the maximum number of multibyte characters to store. Depending on the platform, Unicode characters use either two or four bytes per character and can support all international characters.
REAL	stores a signed, approximate, single-precision, floating-point number.
SMALLINT	stores a small signed, exact whole number, with a precision of five digits. The range of integers is -32,768 to 32,767. Integer data types do not store decimal values; fractional portions are discarded.
TIME(*p*)	stores a time value. A time literal is specified in the format *hh:mm:ss[.nnnnnnnnn]*; a two-digit hour 00 to 23, a two-digit minute 00 to 59, and a two-digit second 00 to 59, with an optional fraction value. For example, the time 6:30 a.m. is specified as `06:30:00`. When supported by a data source, the *p* parameter specifies the seconds precision. The seconds precision is an optional fraction value that is up to nine digits long.
TIMESTAMP(*p*)	stores both date and time values. A timestamp literal is specified in the format *yyyy-mm-dd:hh:mm:ss[.nnnnnnnnn]*: a four-digit year 0001 to 9999, a two-digit month 01 to 12, a two-digit day 01 to 31, a two-digit hour 00 to 23, a two-digit minute 00 to 59, and a two-digit second 00 to 59, with an optional fraction value. For example, the date and time September 24, 1975 6:30 a.m. is specified as `1975-09-24:06:30:00`. When supported by a data source, the *p* parameter specifies the seconds precision. The seconds precision is an optional fraction value that is up to nine digits long.
TINYINT	stores a very small signed, exact whole number, with a precision of three digits. The range of integers is -128 to 127. Integer data types do not store decimal values; fractional portions are discarded.
VARBINARY(*n*)	stores varying-length binary data, where *n* is the maximum number of bytes to store. The maximum number of bytes is not required to store each value. If `varbinary(10)` is specified and the binary string uses only five bytes, only five bytes are stored in the column.

Data Type	Description
VARCHAR(*n*)[1]	stores a varying-length character string, where *n* is the maximum number of characters to store. Each character uses 1 byte of storage. The maximum number of characters is not required to store each value. If `varchar(10)` is specified and the character string is only five characters long, only five characters are stored in the column.

1 The CHAR, NCHAR, NVARCHAR, and VARCHAR data types have a length restriction of 10485760 characters for any column that is used as input or included in the output.

When defining a data type, use the data type keywords for either the data types that are supported by FedSQL or the data types that are supported by the target database SQL language. That is,

- If you submit FedSQL statements, use FedSQL data type keywords.
- If you request explicit SQL pass-through and submit SQL statements using the SQL language that is implemented by the specific data source, use the data type names for the target database. For information about how to define data types using the SQL language for a specific data source, see the documentation for that data source.

Keep in mind that in order for data to be stored, the data type must be available for data storage in that data source. Although FedSQL supports several data types, the data types that can be defined for a particular table depend on the data source, because each data source does not necessarily support all FedSQL data types. In addition, data sources support variations of the standard SQL data types. That is, a specific data type that you specify might map to a different data type and might also have different attributes in the underlying data source. This occurs when a data source does not natively support a specific data type, but data values of a similar data type can be converted without data loss. For example, to support the INTEGER data type, a SAS data set maps the data type definition to SAS numeric, which is a DOUBLE.

For details about data source implementation for each data type, see Appendix 2, “Data Type Reference,” on page 1109.

In addition, the CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source or the specified precision or scale is not within the data source range. For information about the CT_PRESERVE= connection argument, see *SAS Federation Server: Administrator’s Guide*.

Identifiers

Overview of Identifiers

An identifier is one or more tokens, or symbols, that name programming language entities, such as variables, method names, package names, and arrays, as well as data source objects, such as table names and column names.

FedSQL supports ANSI SQL:1999 core standards for both regular and delimited identifiers.

Regular identifiers are the type of identifiers that you see in most programming languages. They are not case-sensitive; the identifier Name is treated the same way as NAME, name, and NaMe: it becomes NAME. All lowercase letters in regular identifiers are changed to uppercase. Only certain characters are allowed in regular identifiers.

Delimited identifiers are case-sensitive and allow any character. Delimited identifiers are enclosed in double quotation marks.

By supporting ANSI SQL:1999 identifiers, FedSQL is compatible with data sources that also support the ANSI SQL:1999 identifiers.

Note: Identifiers for SAS data sets, SPD Engine data sets, and SPD Server tables are limited to 32 characters.

Note: When more than one data source is involved, the maximum length of an identifier is determined by the smallest maximum length that is supported by all of the data sources and FedSQL. For example, if your data sources are a SAS data set, which has a maximum of 32 characters, and MySQL, which has a maximum of 64 characters, the maximum length of an identifier would be 32 characters.

Regular Identifiers

When you name regular identifiers, use these rules:

- The length of a regular identifier can be 1 to 256 characters.
- The first character of a regular identifier must be a letter. Subsequent characters can be letters, digits, or underscores.
- Regular identifiers are case-insensitive.

The following regular identifiers are valid:

```
firstName
```

```
lastName
phone_num1
phone_num2
```

Letters in regular identifiers are stored internally as uppercase letters, which allows letters to be written in any case.

Note: Each data source has its own naming conventions, all of which are accepted by FedSQL. If your program contains identifiers that are specific for a particular data source, you must follow the naming conventions for that data source. See your database documentation for information about its naming conventions.

Delimited Identifiers

When you name delimited identifiers, follow these rules:

- The length of a delimited identifier can be 1 to 256 characters.
- Begin and end delimited identifiers with double quotation marks.
- Delimited identifiers consist of any sequence of characters, including spaces and special characters, between the beginning and ending double quotation marks.
- Delimited identifiers are case-sensitive.

A string of characters enclosed in double quotation marks is interpreted as an identifier and not as a character constant. Character constants can be enclosed only in single quotation marks.

Here is a list of valid delimited identifiers:

```
" x y z"
"Ü1"
"phone_num"
"a & B"
```

Letters in delimited identifiers are case-sensitive and their case is preserved when they are stored in FedSQL. When they are stored, the double quotation marks are removed.

You can use delimited identifiers for terms that might otherwise be a reserved word. For example, to use the term “char” other than for a character declaration, you would use it as the delimited identifier “char”. For more information, see “FedSQL Reserved Words” on page 71.

Note: Each data source has its own naming conventions, all of which are accepted by FedSQL. When your program contains identifiers that are specific for a particular data source, you must follow the naming conventions for that data source. See your database documentation for information about table naming conventions for data source objects.

Referencing a Macro Variable in a Delimited Identifier

To reference a SAS macro variable in a delimited identifier, use the SAS macro function %TSLIT, which overrides the need for double quotation marks around the literal string and puts single quotation marks around the input value.

The %TSLIT macro function is stored in the default autocall macro library. The syntax is as follows:

```
%TSLIT(&macro-name)
```

Note: SAS macro variables are available only to FedSQL programs that are executed in a Base SAS session using PROC FEDSQL. They are not available to FedSQL programs that are executed outside the SAS System, such as on SAS Federation Server. SAS macro variables can be used to pre-process the source before it is prepared and executed. Functions like SYMPUT() and SYMGET() will not work in FedSQL.

Support for Non-Latin Characters

FedSQL supports non-Latin characters only in quoted identifiers. Only Latin characters can be used in non-delimited identifiers.

How FedSQL Processes Nulls and SAS Missing Values

FedSQL Modes for Nonexistent Data

Nonexistent data is represented by a SAS missing value in SAS data sets, SPD Engine data sets, and SPD Server tables. For all other data sources, nonexistent data is represented by an ANSI SQL null value. The SAS missing value indicators, . , ._, .A - .Z, and '_' are known values that indicate nonexistent data. Table data with an ANSI null has no real data value; it is metadata that indicates an unknown value.

Because there are significant differences in processing null values and SAS missing values, FedSQL has two modes for processing nonexistent data: the ANSI SQL null mode (ANSI mode) and the SAS missing value mode (SAS mode).

The behavior of nonexistent data depends on how you connect to the data source:

- By default, a client application that connects to the data source via a client-side driver, such as JDBC or ODBC, processes data using ANSI mode.
- By default, a Base SAS session that submits PROC FEDSQL processes data using SAS mode. PROC FEDSQL provides the ANSIMODE option in order to process data in ANSI mode.

In most instances, no mode change is necessary to process nonexistent data. You might want to change the mode in the following circumstances:

- when a client application processes SAS data sets, SPD Engine data sets, or SPD Server tables and the mode for nonexistent data is in ANSI mode
- when the processing of SAS data sets, SPD Engine data sets, or SPD Server tables is complete and the client application is ready to return to ANSI mode

CAUTION

If the mode is not set for the desired results, data is lost. In ANSI mode, when FedSQL reads a numeric SAS missing data value from the data source, it converts it to a data type of DOUBLE. If the SAS missing data value is a special missing value, such as .A, the .A is lost when it is converted to a null. When a null value is written to a SAS data set, SPD Engine data set, or SPD Server table, FedSQL converts it to a SAS missing value, which is a period (.). In SAS mode, when a null of type CHAR is read from the data source, FedSQL converts it to a blank character. When the blank character is stored in a SAS data set, SPD Engine data set, or SPD Server table, that value can no longer be interpreted as an unknown value in ANSI mode.

For information about how to set FedSQL in ANSI mode and SAS mode, see the SAS documentation for your client environment.

Data Types for SAS Missing Mode

Currently, FedSQL applies SAS missing mode to columns of the DOUBLE and CHAR data types and to comparisons that involve those data types. SAS missing mode is not applied to columns that are defined with the FedSQL DATE, TIME, and TIMESTAMP data types. Those data types are converted to DOUBLE and assigned a SAS format for some data sources. For information about how FedSQL handles date and time values, see “Dates and Times in FedSQL” on page 58. FedSQL treats columns containing date and time values for these sources like DATE, TIME, and TIMESTAMP data types in all other ways.

Differences between Processing Null Values and SAS Missing Values

Processing SAS missing values is different from processing null values and has significant implications in these situations:

- when filtering data (for example, in a WHERE clause, a HAVING clause, or an outer join ON clause). SAS mode interprets null values as SAS missing values,

which are known values, whereas ANSI mode interprets null values as unknown values.

- when submitting outer joins in ANSI mode, internal processing might generate nulls for intermediate result sets. FedSQL might generate SAS missing values in SAS mode for intermediate result sets. Therefore, for intermediate result sets, nulls are interpreted as unknown values in ANSI mode and in SAS mode, missing values are interpreted as known values.
- when comparing a blank character. SAS mode interprets the blank character as a missing value. In ANSI mode, a blank character is a blank character; it has no special meaning.

The following table lists attribute and behavior differences between null values and SAS missing values:

Table 2.2 Attribute and Behavior Differences between Null Values and SAS Missing Values

Attribute or Behavior	Null Values	SAS Missing Values
internal representation	metadata	floating point or character
evaluation by logical operators	is an unknown value that is compared by using three-valued logic, whose resolved values are True, False, and Unknown. For example, `WHERE col1 = null` returns `UNKNOWN`.	is a known value that, when compared, resolves to a Boolean result
collating sequence order	appears as the smallest value	appears as the smallest value

For information about the results of logical operations on null values, see “<search-condition>” on page 982.

Reading and Writing Nonexistent Data in ANSI Mode

Many relational databases such as Oracle and DB2 implement ANSI SQL null values. Therefore, the concept of null values using FedSQL is the same as using the SQL language for databases that support ANSI SQL. It is important to understand how FedSQL processes SAS missing values because data can be lost.

SAS missing value data types can be only DOUBLE or CHAR. Therefore, only the conversion for these data types is shown. The following table shows the value that is returned to the client application when FedSQL reads a null value or a SAS missing value from a data source in ANSI mode:

Table 2.3 Reading Nonexistent Data Values in ANSI Mode

Column Data Type	Nonexistent Data Value	Value Returned to the Application
DOUBLE	., ._, or .A - .Z	null
DOUBLE	null	null
CHAR	'␣'	'␣'
CHAR	null	null

Note: The value '␣' is a blank space between single quotation marks, which, in ANSI mode, is a blank space, not nonexistent data.

This next table shows the value that is stored when nonexistent data values are written to data sources in ANSI mode:

Table 2.4 Writing Nonexistent Data Values in ANSI Mode

Column Data Type	Nonexistent Data Value	Value Stored in a SAS Data Set, SPD Engine Data Set, or SPD Server Table	Value Stored in the ANSI SQL Null Supported Data Source
DOUBLE	., ._, or .A – .Z	.	null
DOUBLE	null	.	null
CHAR	'␣'	'␣'	'␣'
CHAR	null	'␣'	null

Note: The value '␣' is a blank space between single quotation marks, which, in ANSI mode, is a blank space, not nonexistent data.

Reading and Writing Nonexistent Data in SAS Mode

When the client application uses the SAS mode, nonexistent data values are treated like SAS missing values in the Base SAS environment.

The following table shows how nonexistent data values of data type DOUBLE and CHAR are read in SAS mode:

Table 2.5 Reading Nonexistent Data Values in SAS Mode

Column Data Type	Nonexistent Data Value	Value Returned to the Application
DOUBLE	., ._, or .A –.Z	., ._, or .A - .Z
DOUBLE	null	.
CHAR	'␣'	'␣'
CHAR	null	'␣'

Note: The value '␣' is a blank space between single quotation marks, which, in SAS mode, is nonexistent data.

The next table shows how nonexistent data values are written to a data source in SAS mode:

Table 2.6 Writing Nonexistent Data Values in SAS Mode

Column Data Type	Nonexistent Data Value	Value Stored in a SAS Data Set, SPD Engine Data Set, or SPD Server Table	Value Stored in the ANSI SQL Null Supported Data Source
DOUBLE	., ._, or .A - .Z	., ._, or .A - .Z	null
DOUBLE	null	.	null
CHAR	'␣'	'␣'	'␣'
CHAR	null	'␣'	'␣'

Note: The value '␣' is a blank space between single quotation marks, which, in SAS mode, is nonexistent data.

Testing and Modifying Nulls and SAS Missing Values

FedSQL provides the IFNULL function to test for a null value and the NULLIF expression to change a null value.

The IFNULL function takes two expressions as arguments. If the first expression is a null value, it returns the second expression. Otherwise, the function returns the first value:

```
IFNULL(expression, return_value_if_null_expression)
```

If the value of *expression* is null, the function returns the value of *return_value_if_null_expression*.

In this example, all book names are returned for the books that have an unknown value of numCopies:

```
select bookName when ifnull(numCopies, 'T') = 'T';
```

The NULLIF expression also takes two expressions as arguments. If the two expressions are equal, the value that is returned is a null value. Otherwise, the value that is returned is the first SQL expression:

```
NULLIF(expression, test_value_expression);
```

Here, if the value of numCopies is a negative value, –1, it is replaced with a null to indicate an unknown value:

```
update books set numCopies = nullif(numCopies, -1);
```

For more information, see "IFNULL Function" on page 551 and "NULLIF Expression" on page 892.

Type Conversions

Type Conversion Definitions

binary data type
: refers to the VARBINARY and BINARY data type.

character data type
: refers to the CHAR, VARCHAR, NCHAR, and NVARCHAR data types.

coercible data type
: a data type that can be converted to multiple data types, not just a character data type.

date/time data type
: refers to the DATE, TIME, and TIMESTAMP data types.

non-coercible data type
: a data type that can be converted only to a character data type.

numeric data type
: refers to the DECIMAL, NUMERIC, DOUBLE (or FLOAT), REAL, BIGINT, INT, SMALLINT, and TINYINT data types.

standard character conversion
: if an expression is not one of the character data types, it is converted to a CHAR data type.

standard numeric conversion
: if an expression has a coercible, non-numeric data type, it is converted to a DOUBLE data type.

Overview of Type Conversions

Operands in an expression must be of the same general data type — numeric, character, binary, or date/time — in order for FedSQL to resolve the expression. When it is necessary, FedSQL converts an operand's data type to another data type, depending on the operands and operators in the expression. This process is called *type conversion*. A type conversion occurs only if the underlying data source supports it. For example, the concatenation operator (||) operates on character data types. For a database that supports data types INTEGER and CHAR, in a concatenation of the character string “First” and the numeric integer 1, the INTEGER data type for the operand 1 is converted to a CHAR data type before the concatenation takes place.

When an operand data type is converted within the same general data type, the operand data type is promoted. Operands with a data type of SMALLINT and TINYINT are promoted to INTEGER, and operands of type REAL are promoted to DOUBLE. Type promotion is performed for all operations on SMALLINT, TINYINT, and REAL, including arguments for method and function expressions.

Numeric and character data types are coercible. The BINARY, VARBINARY, and the date/time data types DATE, TIME, and TIMESTAMP are non-coercible and can be converted to only one of the character data types by using the PUT function.

When FedSQL evaluates an expression, if the data types of the operands match exactly, no type conversion or promotion is necessary and the expression is resolved. Otherwise, each operand must go through a standard numeric conversion or a standard character conversion, depending on the operator.

The results of a numeric or character expression are based on a data type precedence. If both operands have different types within the same general data type, the data type of the expression is that of the operand with the higher precedence, where 1 is the highest precedence. For example, for numeric data types, a data type of DOUBLE has the highest precedence. If an expression has an operand of type INTEGER and an operand of type DOUBLE, the data type of the expression is DOUBLE. A list of precedences can be found in the topics that follow, if applicable, for the different types of expressions.

Type Conversion for Unary Expressions

In unary expressions, such as `++1` or `-444`, the standard numeric conversion is applied to the operand. The following table shows the data type for unary expressions:

Table 2.7 *Data Type Conversion for Unary Expressions*

Expression Type	Expression Data Type
Unary plus	same as the operand or DOUBLE for converted operands

Expression Type	Expression Data Type
Unary minus	same as the operand or DOUBLE for converted operands
Unary not	INTEGER

Type Conversion for Logical Expressions

In logical expressions, such as `(a <> start) OR (f = finish)`, the standard numeric conversion is applied to each operand. The following table shows the precedence that is used to determine the data type of the expression, where 1 is the highest precedence and 3 is the lowest. The data type of the expression is the data type of the operand that has the higher precedence.

Table 2.8 *Data Type Conversion for Logical Expressions*

Precedence	Data Type of Either Operand	Expression Data Type
1	DOUBLE	DOUBLE
2	BIGINT	BIGINT
3	all other numeric data types	INTEGER

Type Conversion for Arithmetic Expressions

In arithmetic expressions, such as `a<>b` or `a + (b * c)`, the standard numeric conversion is applied to each operand.

The following table shows the precedence that is used to determine the data type of arithmetic expressions for the addition, subtraction, multiplication, and division operators, where 1 is the highest precedence and 3 is the lowest. The data type of the expression is the data type of the operand that has the higher precedence.

Table 2.9 *Type Conversion for Addition, Subtraction, Multiplication, and Division Expressions*

Precedence	Data Type of Either Operand	Expression Data Type
1	DOUBLE	DOUBLE

Precedence	Data Type of Either Operand	Expression Data Type
2	BIGINT	BIGINT
3	all other numeric data types	INTEGER

The following table shows the data type for arithmetic expressions that use the power operator:

Table 2.10 Data Type Conversion for the Min, Max, and Power Operator Expressions

Operator	Operator Data Type	Expression Data Type
**	all numeric data types	DOUBLE

Type Conversion for Relational Expressions

In relational expressions, such as `x <= y` or `i > 4`, the standard conversion that is applied depends on the operand data types. The data type of the expression is always BOOLEAN, as shown in the following tables.

Table 2.11 Data Type Conversion for Relational Expressions except IN Expressions

Order of Data Type Resolution	Data Type of Either Operand	Standard Conversion	Expression Data Type
1	any numeric data type	numeric	BOOLEAN
2	CHAR/NCHAR	character	BOOLEAN
3	DATE, TIME, TIMESTAMP	none, data types must match	BOOLEAN
4	all other data types	none, error returned	not applicable

Table 2.12 Data Type Conversion for IN Expressions

Operand	Operand Conversion	Expression Data Type
all	standard numeric or standard character	BOOLEAN

Type Conversion for Concatenation Expressions

In concatenation expressions, such as `a || b` or `x !! y`, the standard character conversion is applied to each operand. The following table shows the precedence used to determine the data type of the expression, where 1 is the highest precedence and 2 is the lowest. The data type of the expression is the data type of the operand that has the higher precedence.

Table 2.13 *Data Type Conversion for Concatenation Expressions*

Precedence	Data Type of Either Operand	Expression Data Type
1	if either is type NCHAR	NCHAR
2	CHAR	CHAR

NLS Transcoding Failures

Transcoding is the process of converting character data from one encoding to another encoding. An NLS transcoding failure can occur during row input or output operations, or during string assignment. By default, this run-time error causes row processing to halt. You can change the default behavior by using one of the following options:

- SAS Federation Server: specify the DEFAULT_ATTR= connection option with the XCODE_WARN=*n* statement handle option.
- PROC FEDSQL and PROC DS2: set the XCODE= option.

Using the options, you can choose to ignore the errors and continue processing of the row.

For more information, see the SAS Federation Server and SAS procedure documentation.

Join Operations

Overview of Join Operations

A join operation is a query that combines data from two or more tables or views based usually on relationships among the data in those tables. When multiple table specifications are listed in the FROM clause of a SELECT statement, they are processed to form one result set. The result set contains data from each contributing table and can be saved as a table or used as is. Most join operations contain at least one join condition, which is either in the FROM clause or in a WHERE clause.

For example, you can join the data of two tables based on the values of a column that exists in both tables. The following query joins the two tables Products and Sales. FedSQL creates the result set by retrieving the data for columns Product and Totals where the values match for the column Prodid.

```
select products.product, sales.totals
   from products, sales
   where products.prodid=sales.prodid;
```

Output 2.1 *Join Result Sets of Tables Products and Sales*

PRODUCT	TOTALS
Wheat	$189,400
Rice	$555,789
Corn	$781,183
Corn	$2,789,654

Most joins are of two tables. However, you can join more than two tables. To perform a join operation of three or more tables, FedSQL first joins two tables based on the join condition. Then FedSQL joins the results to another table based on the join condition. This process continues until all tables are joined into the result set. The following query first joins tables Products and Sales, which produces a result set, and then joins the result set and the table Customers, which produces the final result set.

```
select products.product, sales.totals, customers.city
   from products, sales, customers
   where products.prodid=sales.prodid and sales.custid=customers.custid;
```

Output 2.2 *Join Result Set of Tables Products, Sales, and Customers*

PRODUCT	TOTALS	CITY
Wheat	$189,400	Boulder
Rice	$555,789	Nagasaki
Corn	$781,183	Tokyo
Corn	$2,789,654	Little Rock

FedSQL supports several join operations such as simple joins, equijoins, cross joins, qualified joins, and natural joins. Appropriate syntax determines the type of join operation. In addition, the qualified and natural join operations can be affected by specifying the join type, which can be an inner join or an outer join.

Note: The join operation examples in this section use the tables Customers, Products, and Sales. To view the tables, see Appendix 4, “Tables Used in Examples,” on page 1169.

Understanding the Join Operations

Simple Join

A simple join is the most basic type of join where multiple tables, separated by commas, are listed in the FROM clause of a SELECT statement. There is no join condition. Joining tables in this way produces a result set where each row from the first table is combined with each row of the second table, and so on.

This simple join example selects all columns and all rows from the tables Products and Sales.

```
select * from products, sales;
```

Output 2.3 *Simple Join of Two Tables*

PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
1424	Rice	3234	1	$189,400	United States
3421	Corn	3234	1	$189,400	United States
3234	Wheat	3234	1	$189,400	United States
3485	Oat	3234	1	$189,400	United States
1424	Rice	1424	3	$555,789	Japan
3421	Corn	1424	3	$555,789	Japan
3234	Wheat	1424	3	$555,789	Japan
3485	Oat	1424	3	$555,789	Japan
1424	Rice	3421	4	$781,183	Japan
3421	Corn	3421	4	$781,183	Japan
3234	Wheat	3421	4	$781,183	Japan
3485	Oat	3421	4	$781,183	Japan
1424	Rice	3421	2	$2,789,654	United States
3421	Corn	3421	2	$2,789,654	United States
3234	Wheat	3421	2	$2,789,654	United States
3485	Oat	3421	2	$2,789,654	United States
1424	Rice	3975	5	$899,453	Argentina
3421	Corn	3975	5	$899,453	Argentina
3234	Wheat	3975	5	$899,453	Argentina
3485	Oat	3975	5	$899,453	Argentina

This example is also a simple join, but the SELECT statement specifies one column from each of three tables. Each row from the first table is combined with each row from the second table, which are then combined with each row from the third table. The result is a large, basically meaningless result set. The following output shows only a portion of the result set.

```
select products.product, sales.totals, customers.country
   from products, sales, customers;
```

Output 2.4 *Simple Join of Three Tables*

PRODUCT	TOTALS	COUNTRY
Rice	$189,400	United States
Corn	$189,400	United States
Wheat	$189,400	United States
Oat	$189,400	United States
Rice	$555,789	United States
Corn	$555,789	United States
Wheat	$555,789	United States
Oat	$555,789	United States
Rice	$781,183	United States
Corn	$781,183	United States
Wheat	$781,183	United States
Oat	$781,183	United States
Rice	$2,789,654	United States
Corn	$2,789,654	United States
Wheat	$2,789,654	United States

Equijoin

An equijoin is a simple join that is subset with a WHERE clause. The join condition must be an equality comparison. An equijoin produces a more meaningful result than just a simple join, because only rows meeting the equality test are returned. Multiple match criteria can be specified by using the AND operator. When multiple match criteria are specified, only rows that meet all of the equality tests are returned.

This equijoin example selects all columns from the tables Products and Sales where the values match for the column Prodid, which exists in both tables. Because all columns are selected with the * notation, the Prodid column is duplicated in the result set.

```
select * from products, sales
   where products.prodid=sales.prodid;
```

Output 2.5 *Equijoin of All Columns*

PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
3234	Wheat	3234	1	$189,400	United States
1424	Rice	1424	3	$555,789	Japan
3421	Corn	3421	4	$781,183	Japan
3421	Corn	3421	2	$2,789,654	United States

When you specify the columns Prodid, Product, and Totals in the SELECT statement, the column Prodid is not duplicated, even though it exists in both the Products and Sales tables. The result set includes the data where the values match for the column Prodid.

```
select products.prodid, products.product, sales.totals
   from products, sales
   where products.prodid=sales.prodid;
```

Output 2.6 *Equijoin with Specified Columns*

PRODID	PRODUCT	TOTALS
3234	Wheat	$189,400
1424	Rice	$555,789
3421	Corn	$781,183
3421	Corn	$2,789,654

This equijoin example selects the columns Product, Totals, and City from the tables Products, Sales, and Customers. First, the result set includes the data where the values match for the column Prodid, which exists in tables Products and Sales. Then the result set is combined with the data where the values match for the column Custid, which exists in tables Sales and Customers.

```
select products.product, sales.totals, customers.city
   from products, sales, customers
   where products.prodid=sales.prodid and sales.custid=customers.custid;
```

Output 2.7 *Equijoin of Three Tables*

PRODUCT	TOTALS	CITY
Wheat	$189,400	Boulder
Rice	$555,789	Nagasaki
Corn	$781,183	Tokyo
Corn	$2,789,654	Little Rock

Cross Join

A cross join is a relational join that results in a Cartesian product of two tables. A cross join is requested with the syntax CROSS JOIN. A cross join can be subset with a WHERE clause, but you cannot use an ON clause.

This cross join example selects all columns and all rows from the tables Products and Sales, which produces the same results as a simple join of two tables.

```
select * from products cross join sales;
```

Output 2.8 *Cross Join of Two Tables*

PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
1424	Rice	3234	1	$189,400	United States
3421	Corn	3234	1	$189,400	United States
3234	Wheat	3234	1	$189,400	United States
3485	Oat	3234	1	$189,400	United States
1424	Rice	1424	3	$555,789	Japan
3421	Corn	1424	3	$555,789	Japan
3234	Wheat	1424	3	$555,789	Japan
3485	Oat	1424	3	$555,789	Japan
1424	Rice	3421	4	$781,183	Japan
3421	Corn	3421	4	$781,183	Japan
3234	Wheat	3421	4	$781,183	Japan
3485	Oat	3421	4	$781,183	Japan
1424	Rice	3421	2	$2,789,654	United States
3421	Corn	3421	2	$2,789,654	United States
3234	Wheat	3421	2	$2,789,654	United States
3485	Oat	3421	2	$2,789,654	United States
1424	Rice	3975	5	$899,453	Argentina
3421	Corn	3975	5	$899,453	Argentina
3234	Wheat	3975	5	$899,453	Argentina
3485	Oat	3975	5	$899,453	Argentina

This cross join example selects the columns Prodid, Product, and Totals from tables Products and Sales. The result set includes the data where the values match for the column Prodid. The results are the same as an equijoin of two tables.

```
select products.prodid, products.product, sales.totals
   from products cross join sales
   where products.prodid=sales.prodid;
```

Output 2.9 *Cross Join with a WHERE Clause*

PRODID	PRODUCT	TOTALS
3234	Wheat	$189,400
1424	Rice	$555,789
3421	Corn	$781,183
3421	Corn	$2,789,654

Qualified Join

A qualified join provides an easy way to control which rows appear in the result set. You can use any columns to match rows from one table against those from another table. A qualified join is requested with the syntax JOIN and then the syntax ON or USING to specify the join condition. You can use a WHERE clause to further subset the query results.

- The ON clause specifies a join condition to filter the data. The ON clause accepts search conditions such as conditional expressions like the WHERE clause. The ON clause joins tables where the column names do not match in both tables. For columns that exist in both tables, the ON clause preserves the columns from each joined table separately in the result set.
- The USING clause specifies columns to test for equality. The columns listed in the USING clause must be present in both tables. The USING clause is like a shorthand way of defining join conditions without having to specify a qualifier. The USING clause is equivalent to a join condition where each column from the left table is compared to a column with the same name in the right table. For columns that exist in both tables, the USING clause merges the columns from the joined tables into a single column.

A qualified join can be an inner join or an outer join, which is requested with the syntax INNER or OUTER. If the join type specification is omitted, then an inner join is implied. See “Inner and Outer Join Types” on page 40.

This qualified join example selects all columns from the tables Products and Sales. The returned rows are filtered based on the column Country in the Sales table, where the value in Country equals United States. The column Prodid exists in both tables and is duplicated in the result set.

```
select * from products join sales
   on (sales.country='United States');
```

Output 2.10 *Qualified Join with an ON Clause*

PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
1424	Rice	3234	1	$189,400	United States
3421	Corn	3234	1	$189,400	United States
3234	Wheat	3234	1	$189,400	United States
3485	Oat	3234	1	$189,400	United States
1424	Rice	3421	2	$2,789,654	United States
3421	Corn	3421	2	$2,789,654	United States
3234	Wheat	3421	2	$2,789,654	United States
3485	Oat	3421	2	$2,789,654	United States

This qualified join example selects all columns from the tables Products and Sales. The returned rows are filtered by selecting the values that match for the column Prodid, which exists in both tables. The USING clause is like a shorthand way of defining join conditions without having to specify a qualifier. The USING clause is equivalent to a join condition where each column from the left table is compared to a column with the same name in the right table. Unlike an equijoin and a cross join, the column Prodid is not duplicated in the result set.

```
select * from products join sales
   using (prodid);
```

Output 2.11 *Qualified Join with a USING Clause*

PRODID	PRODUCT	CUSTID	TOTALS	COUNTRY
3234	Wheat	1	$189,400	United States
1424	Rice	3	$555,789	Japan
3421	Corn	4	$781,183	Japan
3421	Corn	2	$2,789,654	United States

This qualified join example selects columns Prodid, Product, and Totals from the tables Products and Sales. The returned rows are filtered based on the column Country where the value equals United States. The returned rows are further subset where the value for Product equals Rice.

```
select products.prodid, products.product, sales.totals
   from products join sales
   on (sales.country='United States')
   where products.product='Rice';
```

Output 2.12 *Qualified Join with an ON Clause and a WHERE Clause*

PRODID	PRODUCT	TOTALS
1424	Rice	$189,400
1424	Rice	$2,789,654

Natural Join

A natural join selects rows from two tables that have equal values in columns that share the same name and the same type. A natural join is requested with the syntax NATURAL JOIN. If like columns are not found, then a cross join is performed. Do not use an ON clause with a natural join. When using a natural join, an ON clause is implied, matching all like columns. You can use a WHERE clause to subset the query results. A natural join functions the same as a qualified join with the USING clause. A natural join is a shorthand of USING. Like USING, like columns appear only once in the result set.

A natural join can be an inner join or an outer join, which is requested with the syntax INNER or OUTER. If the join type specification is omitted, then an inner join is implied. See "Inner and Outer Join Types" on page 40.

This natural join example selects all columns from the tables Products and Sales. The result set includes the data where the values match for the column Prodid, which exists in both tables. Unlike a cross join and a simple join of two tables, the natural join result set does not include duplicate Prodid columns.

```
select * from products natural join sales;
```

Output 2.13 *Natural Join of All Columns*

PRODID	PRODUCT	CUSTID	TOTALS	COUNTRY
3234	Wheat	1	$189,400	United States
1424	Rice	3	$555,789	Japan
3421	Corn	4	$781,183	Japan
3421	Corn	2	$2,789,654	United States

This natural join example selects columns City and Totals from the tables Sales and Customers. The result set includes the data where the values match for the columns Custid and Country, which exist in both tables. The returned rows are subset where the value for Country equals United States.

```
select customers.city, sales.totals
   from sales natural join customers
   where customers.country='United States';
```

Output 2.14 *Natural Join with a WHERE Clause*

CITY	TOTALS
Boulder	$189,400
Little Rock	$2,789,654

Inner and Outer Join Types

Overview of Inner and Outer Join Types

The result set from a qualified join and a natural join can be affected by specifying the join type, which can be an inner join or an outer join. By default, qualified joins and natural joins function as inner joins.

Inner Joins

An inner join returns a result set that includes all rows from the first table that matches rows from the second table. Inner joins return only those rows that satisfy the join condition. Unmatched rows from both tables are discarded. By default, qualified joins and natural joins function as inner joins. Including the syntax INNER has no additional effects on the result set.

```
select * from products inner join sales
   on (sales.country='United States');
```

```
select customers.city, sales.totals
   from sales natural inner join customers
   where country='United States';
```

Outer Joins

An outer join returns a result set that includes all rows that satisfy the join condition as well as unmatched rows from one or both tables. An outer join can be a left, right, or full outer join. An inner join discards any rows where the join condition is not met, but an outer joins maintains some or all of the unmatched rows.

For an outer join, a specified WHERE clause is applied after the join is performed and eliminates all rows that do not satisfy the WHERE clause. Applying a WHERE clause to an outer join can sometimes defeat the purpose, because the WHERE clause deletes the very rows that the outer join retains.

- A left outer join preserves unmatched rows from the left table, which is the first table listed in the SELECT statement. A left outer join returns a result set that includes all rows that satisfy the join condition and rows from the left table that do not match the join condition. Therefore, a left outer join returns all rows from the left table, and only the matching rows from the right table. A left outer join is requested with the syntax LEFT [OUTER].

  This qualified join example returns a result set that includes all rows from both tables that satisfy the join condition. The join condition filters rows based on the column Country where the value equals United States. The result set also includes rows from the Customers table that do not match the join condition. As a left outer join, all rows from the Customers table are returned.

```
select customers.city, sales.totals
   from customers left outer join sales
   on (customers.country='United States');
```

CITY	TOTALS
Boulder	$189,400
Little Rock	$189,400
Boulder	$555,789
Little Rock	$555,789
Boulder	$781,183
Little Rock	$781,183
Boulder	$2,789,654
Little Rock	$2,789,654
Boulder	$899,453
Little Rock	$899,453
Nagasaki	.
Tokyo	.
Buenos Aires	.

  This natural join example returns a result set that includes all rows from both tables that satisfy the join condition, which includes the data where the values match for the column Prodid The result set also includes a row from the Sales table that does not match the join condition. As a left outer join, all rows from the Sales table are returned.

```
select * from sales natural left outer join products;
```

PRODID	CUSTID	TOTALS	COUNTRY	PRODUCT
3234	1	$189,400	United States	Wheat
1424	3	$555,789	Japan	Rice
3421	4	$781,183	Japan	Corn
3421	2	$2,789,654	United States	Corn
3975	5	$899,453	Argentina	

- A right outer join preserves unmatched rows from the right table, which is the second table listed in the SELECT statement. A right outer join returns a result set that includes all rows that satisfy the join condition and rows from the right table that do not match the join condition. Therefore, a right outer join returns all rows from the right table, and only the matching rows from the left table. A right outer join is requested with the syntax RIGHT [OUTER].

  This qualified join example returns a result set that includes all rows from both tables that satisfy the join condition. The join condition filters rows based on the column Country where the value equals United States. The result set also includes rows from the Sales table that do not match the join condition. As a right outer join, all rows from the Sales table are returned.

```
select * from products right outer join sales
   on (sales.country='United States');
```

PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
1424	Rice	3234	1	$189,400	United States
1424	Rice	3421	2	$2,789,654	United States
3421	Corn	3234	1	$189,400	United States
3421	Corn	3421	2	$2,789,654	United States
3234	Wheat	3234	1	$189,400	United States
3234	Wheat	3421	2	$2,789,654	United States
3485	Oat	3234	1	$189,400	United States
3485	Oat	3421	2	$2,789,654	United States
.		1424	3	$555,789	Japan
.		3421	4	$781,183	Japan
.		3975	5	$899,453	Argentina

This natural join example returns a result set that includes all rows from both tables that satisfy the join condition, which includes the data where the values match for the column Prodid. The result set also includes a row from the Sales table that does not match the join condition. As a right outer join, all rows from the Sales table are returned.

```
select * from products natural right outer join sales;
```

PRODID	PRODUCT	CUSTID	TOTALS	COUNTRY
3234	Wheat	1	$189,400	United States
1424	Rice	3	$555,789	Japan
3421	Corn	4	$781,183	Japan
3421	Corn	2	$2,789,654	United States
3975		5	$899,453	Argentina

- A full outer join preserves unmatched rows from both tables. That is, a full outer join returns all matching and unmatching rows from the left and right table. A full outer join is requested with the syntax FULL [OUTER].

  This qualified join example returns a result set that includes all rows from both tables that satisfy the join condition. The join condition filters rows based on the column Product containing the value Rice. The result set also includes all rows from both tables that do not match the join condition. As a full outer join, all rows from both tables are returned.

```
select * from products full outer join sales
   on (products.product='Rice');
```

PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
1424	Rice	3234	1	$189,400	United States
1424	Rice	1424	3	$555,789	Japan
1424	Rice	3421	4	$781,183	Japan
1424	Rice	3421	2	$2,789,654	United States
1424	Rice	3975	5	$899,453	Argentina
3421	Corn	.	.	.	
3234	Wheat	.	.	.	
3485	Oat	.	.	.	

  This natural join example returns a result set that includes all rows from both tables that satisfy the join condition, which includes the data where the values match for the column Prodid. The result set also includes a row from the Sales table and a row from the Products table that does not match the join condition. As a full outer join, all rows from both tables are returned.

```
select * from products natural full outer join sales;
```

PRODID	PRODUCT	CUSTID	TOTALS	COUNTRY
3234	Wheat	1	$189,400	United States
1424	Rice	3	$555,789	Japan
3421	Corn	4	$781,183	Japan
3421	Corn	2	$2,789,654	United States
3485	Oat	.	.	
3975		5	$899,453	Argentina

Joining Heterogeneous Data

Because typical organizations store data in multiple databases, FedSQL supports joining heterogeneous data. A heterogeneous join occurs when the tables in a join operation exist on different data sources. A heterogeneous join is one type of a federated query.

All FedSQL join operations can be performed as heterogeneous joins. For example, the following natural join is a heterogeneous join between an Oracle table and a Teradata table.

```
select * from oracle.product natural left outer join tera.sales;
```

This heterogeneous join example queries both a SAS data set and an Oracle table. PROC FEDSQL uses the attributes of the librefs to connect to the two data sources. The SAS data set MyBase.Products contains the columns Prodid and Product. The Oracle table Oracle.Sales contains the columns Prodid, Totals, and Country. The SELECT statement joins the columns and rows from both tables based on the column Prodid, which exists in both tables.

```
libname mybase base 'C:\Base';
libname myoracle oracle user=scott password=tiger path=oraclev11;

proc fedsql;
   select mybase.products.prodid, mybase.products.product,
myoracle.sales.totals
      from mybase.products, myoracle.sales
      where products.prodid=sales.prodid;
quit;
```

Output 2.15 *FedSQL Heterogeneous Join of a SAS Data Set and an Oracle Table*

PRODID	PRODUCT	TOTALS
3234	Wheat	$189,400
1424	Rice	$555,789
3421	Corn	$781,183
3421	Corn	$2,789,654

This heterogeneous join example queries three tables: two Oracle tables and one SAS data set. In the query, the join of MyOracle.Products and MyOracle.Customers is performed by Oracle. The join of the Oracle result set with the SAS data set MyBase.Sales is performed by FedSQL.

```
libname mybase base 'C:\Base';
libname myoracle oracle user=scott password=tiger path=oraclev11;

proc fedsql;
   select myoracle.products.product, mybase.sales.totals,
myoracle.customers.city
      from myoracle.products, mybase.sales, myoracle.customers
      where products.prodid=sales.prodid and
sales.custid=customers.custid;
quit;
```

Output 2.16 *FedSQL Heterogeneous Join of Two Oracle Tables and a SAS Data Set*

PRODUCT	TOTALS	CITY
Wheat	$189,400	Boulder
Rice	$555,789	Nagasaki
Corn	$781,183	Tokyo
Corn	$2,789,654	Little Rock

FedSQL Expressions

Query Expressions and Subqueries

Overview of Query Expressions

A *query expression* or *query* consists of one or more SELECT statements that produce a result set. One way that multiple SELECT statements can be combined is by using subqueries. Another is by using set operators. This section discusses set operators. For information about subqueries, see "Overview of Subqueries" on page 51.

FedSQL supports UNION, EXCEPT, and INTERSECT set operators.

- The UNION set operator combines the result of two queries into a single result set that contains the unique rows from both queries.
- The EXCEPT operator combines the result of two queries into a single result set that contains unique rows from the first query.
- The INTERSECT operator combines the result of two queries into a single result set that contains unique instances of rows that are common to both queries.

By default, columns are combined by position in the input tables, regardless of the column names. Columns in the same relative position in the two queries must have compatible data types and lengths and each query must have the same number of columns for the set operation to succeed. The result set contains only the unique rows that result from a query expression. That is, the set operators automatically eliminate duplicate rows from the result sets.

The set operators support the optional keywords to change the default behavior.

Table 2.14 *Optional Keywords for the UNION, EXCEPT, and INTERSECT Set Operators*

Keyword	Purpose
ALL	Returns all rows from both queries, including duplicates, instead of only unique rows.
CORRESPONDING	Combines columns from the queries by column name instead of by position. Columns with non-matching column names are ignored. The CORRESPONDING keyword returns rows for all columns that have matching names.
CORRESPONDING and BY	Combines columns by column name and returns only columns from a column list.

A query expression with set operators is evaluated as follows:

- Each SELECT statement is evaluated to produce a virtual, intermediate result set.
- Each intermediate result set then becomes an operand that is linked with a set operator to form an expression (for example, A UNION B). (When the CORRESPONDING and BY keywords are specified together, columns that are not in the column list are dropped.)
- If the query expression involves more than two SELECT statements, then the result from the first two queries becomes an operand for the next set operator and operand, such as (A UNION B) EXCEPT C, ((A UNION B) EXCEPT C) INTERSECT D, and so on.
- Evaluating a query expression produces a single output result set.

Default Query Expression Behavior

To understand how query expressions work, consider the following examples. The examples operate on two tables, named Numbers1 and Numbers2. The following figures show the content of the tables. For information about creating the tables, see Appendix 4, “Tables Used in Examples,” on page 1169.

Figure 2.1 *Content of Numbers1 Table*

X	Y	Z
one	1	10
three	3	30
four	4	40
four	4	40
five	5	50
five	5	50

Figure 2.2 *Content of Numbers2 Table*

X	Y	Z
four	4	40
five	5	50
five	5	50
one	1	10
two	2	20

Note that the columns in these tables are of the same type and in the same position: X is a character column; Y and Z are numeric columns.

The following example code specifies the UNION operator:

```
select * from numbers1 union select * from numbers2;
```

The UNION operator returns the unique rows from both tables to display as a single, virtual table.

Output 2.17 *Output from the UNION Operation*

X	Y	Z
five	5	50
four	4	40
one	1	10
three	3	30
two	2	20

The following example code specifies the EXCEPT operator:

```
select * from numbers1 except select * from numbers2;
```

The EXCEPT operator returns unique rows that exist in the first table, but not the second table.

Output 2.18 *Output from the EXCEPT Operation*

X	Y	Z
three	3	30

The following example code specifies the INTERSECT operator:

```
select * from numbers1 intersect select * from numbers2;
```

The INTERSECT operator returns unique instances of rows that the specified tables have in common.

Output 2.19 *Output from the INTERSECT Operation*

X	Y	Z
five	5	50
four	4	40
one	1	10

Now consider a third table, Numbers3.

Figure 2.3 Content of Numbers3 Table

Z	Y	X
40	4	four
60	6	six
70	7	seven

Table Numbers3 has columns of the same name and type as tables Numbers1 and Numbers2, but the columns are in a different order. A request to combine table Numbers1 or Numbers2 with table Numbers3 will fail with `ERROR: types CHAR and DOUBLE cannot be matched`.

Combining Queries with the CORRESPONDING Keyword

The CORRESPONDING keyword combines columns from input tables by column name, so the columns do not have to be in the same position. The matching columns must also be of compatible data types. For example, rows from a column of type VARCHAR can be combined with rows from a column of type CHAR. The input tables are not required to have the same number of columns.

Here is an example of a UNION set operation that uses the CORRESPONDING keyword. The request combines rows from tables Numbers1 and Numbers3, which were introduced in "Default Query Expression Behavior" on page 47.:

```
select * from Numbers1 UNION CORRESPONDING select * from Numbers3;
```

Output 2.20 Output from UNION with the CORRESPONDING Keyword

X	Y	Z
five	5	50
four	4	40
one	1	10
seven	7	70
six	6	60
three	3	30

The UNION CORRESPONDING operation returns unique rows from both columns. The columns in the result set use the column ordering of the first query in the expression.

Here is an example of an EXCEPT set operation on the same tables that uses the CORRESPONDING keyword.

```
select * from Numbers1 EXCEPT CORRESPONDING select * from Numbers3;
```

Output 2.21 *Output from the EXCEPT Operation with the CORRESPONDING Keyword*

X	Y	Z
five	5	50
one	1	10
three	3	30

The EXCEPT CORRESPONDING operation returns rows that are unique to the first query, again using the column ordering from the first query.

Here is an example of an INTERSECT set operation on tables Numbers1 and Number3 that specifies the CORRESPONDING keyword.

```
select * from Numbers1 INTERSECT CORRESPONDING select * from Numbers3;
```

Output 2.22 *Output from the INTERSECT Operation with the CORRESPONDING Keyword*

X	Y	Z
four	4	40

The result set includes an instance of the unique row that the two input tables have in common.

Here is an example of an INTERSECT CORRESPONDING BY set operation that also specifies the BY keyword. The code requests values for column Y.

```
select * from Numbers1 INTERSECT CORRESPONDING BY (Y) select * from
Numbers3;
```

Output 2.23 *Output from INTERSECT with the CORRESPONDING and BY Keywords*

Y
4

Combining Set Operators

Here is an example that illustrates how the set operators can be used together.

```
select * from numbers1
intersect select * from numbers2
union corresponding select * from numbers3;
```

The first query expression selects the unique instances of rows that are common to tables Numbers1 and Numbers2. The second query expression merges unique rows from table Numbers3 with the intermediate result set. The CORRESPONDING keyword enables table Numbers3 to be evaluated against the intermediate the result set.

Overview of Subqueries

A *subquery* is a query expression that is nested as part of another query expression. It is specified within parenthesis and has the purpose of returning a value. A subquery can return atomic values (one column with one row in it – also known as a *scalar query*), row values (one row for one or many columns), or table values (one or many rows for one or many columns).

A subquery can be used in the SELECT, INSERT, UPDATE, and DELETE statements. The purpose of a subquery is to enable the contents of one table to influence a query or an action on another table.

Subqueries can appear in various places within a query:

- SELECT Statement
- WHERE Clause
- HAVING Clause
- FROM Clause

Scalar subqueries can be specified in all four locations, anywhere a scalar value can be used. Subqueries that return row values are typically specified in the WHERE clause. Subqueries that return table values are specified in the FROM clause.

A subquery can be dependent or independent of the outer query. When the information pursued in a subquery is dependent in some way on data known to the outer query, we say that the data is correlated with the outer query. A *correlated subquery* typically references the data in the outer query with a correlation name or uses the EXISTS or IN predicate, and uses data from the outer query. The information retrieved by the correlated subquery will change if the data processed by the outer query changes. A correlated subquery is evaluated for each row identified by the outer query, making the subquery resource-intensive. Many correlated queries can be restated in terms of a join operation.

A subquery that is not dependent on the outer query is referred to as a non-correlated query. A non-correlated subquery does not interact much with the data being accumulated in the rest of the query. The non-correlated subquery is evaluated just once and the result used repeatedly in the evaluation of an outer query. Most importantly, the result of the subquery does not change if the data processed by the outer query changes.

Subqueries can be nested. If more than one subquery is used in a query expression, then the innermost query is evaluated first, then the next innermost query, and so on, moving outward.

Examples of Correlated Subqueries

Here is an example of a correlated subquery that specifies the EXISTS predicate. It uses data from the example Employees and Depts tables. For information to create these tables, see “Employees” on page 1175 and “Depts” on page 1174.

```
select *
from employees e
```

```
where exists(select * from depts d
             where d.deptno = e.dept
             and e.pos <> 'Manager');
```

The EXISTS predicate stipulates to return information from the table in the outer query only for values that also exist in the inner query. Notice the second WHERE clause uses the column reference *e.dept*. If you look at the FROM clause in the outer query, you will see that *e* is a correlation name associated with the Employees table that is being used in the outer query.

Output 2.24 *Output of Correlated Query with EXISTS Predicate*

EMPID	DEPT	EMP_NAME	POS	HIRE_DATE
5	20	Greg Welty	Developer	26NOV2001
6	20	Penny Jackson	Developer	26NOV2004
7	10	Edward Murray	Sales Associate	26NOV2001
8	10	Ronald Thomas	Sales Associate	26NOV2002
9	30	Elsie Marks	Executive Assistant	11FEB2002
10	40	Bruno Kramer	Grounds support technician	02NOV2003

Here is a more efficient way to write this query:

```
select *
from employees
where dept in (select deptno from depts
               where pos <> 'Manager');
```

This query does not require a correlation name. The contents of the Dept and Deptno columns are compared with the IN predicate. The query returns the same result as the previous query. Meanwhile, the inner query can be executed once and its results compared to each row in the Employees table.

Examples of Scalar Subqueries

A scalar subquery returns one value for one column. As such, it is useful for aggregate queries.

Here is an example of a scalar subquery in the HAVING clause. The example uses the Employees and Depts tables that were used in "Examples of Correlated Subqueries" on page 51.

```
select dept, count(emp_name)
from mybase.employees e
group by dept
having dept in (select deptno from mybase.depts);
```

It is a simple request: count the number of employees in the Employees table, group the count by department, and return information only about departments having a matching value in the Deptno column of the Depts table.

Output 2.25 *Number of Employees by Department*

DEPT	COUNT
10	3
20	3
30	2
40	2

Here is an example of a scalar subquery that is specified in the SELECT statement.

```
select d.deptno,
       d.deptname,
       d.manager,
      (select count (*) from employees e
       where e.dept = d.deptno and
       e.pos <> 'Manager') as Employees
       from depts d;
```

The subquery counts the number of records in the Employees table, removes employee records that belong to managers, and then returns an aggregate value for each department code that has a match in both tables. That is, one value is returned for each department. It is also a correlated subquery.

Output 2.26 *Number of Employees by Manager*

DEPTNO	DEPTNAME	MANAGER	EMPLOYEES
10	Sales	Jim Barnes	2
20	Research	Clifford James	2
30	Accounting	Barbara Sandman	1
40	Operations	William Baylor	1

Here is an example of a scalar subquery in the WHERE clause:

```
select *
from depts d
where (select COUNT(*) FROM employees e
       WHERE d.deptno = e.dept
       and e.pos <> 'Manager') > 1;
```

As in the previous example, the subquery counts all of the rows in Employees table minus those that describe a manager and correlates them to the DeptNo column of the Depts table. However, the WHERE clause further qualifies the query to retrieve information only about departments that have more than one employee.

Output 2.27 *Departments with More Than One Employee*

DEPTNO	DEPTNAME	MANAGER
10	Sales	Jim Barnes
20	Research	Clifford James

Here is an example of a scalar subquery in the INSERT statement. This example uses data from the Densities example table, which has a column named Population. A new table, Summary, is created and populated with aggregated values from Densities table's Population column. For more information about the Densities table, see "Densities" on page 1173.

```
create table summary (
  sum_pop double having format comma12.,
  max_pop double having format comma12.,
  min_pop double having format comma12.,
  avg_pop double having format comma12.
);

insert into summary (
    sum_pop,
    max_pop,
    min_pop,
    avg_pop)
values (
    (select sum(population) from densities),
    (select max(population) from densities),
    (select min(population) from densities),
    (select avg(population) from densities)
);
select * from summary;
```

Each subquery in the INSERT statement is a scalar subquery. Each subquery returns one value (the sum, maximum, minimum, and average values) from the data in one column, Population, to populate the new table.

Output 2.28 *Content of the Summary Table*

SUM_POP	MAX_POP	MIN_POP	AVG_POP
245,550,884	34,248,705	64,634	12,277,544

Examples of Non-Correlated Queries

A non-correlated subquery is executed before the outer query and the subquery is executed only once. The data from the outer query and the subquery are independent and one execution of the subquery will work for all the rows from the outer query.

The examples in this section use data from the following three tables.

Output 2.29 *Table All1*

x	c
1	one
2	two
3	three
4	four
5	five
6	six

Output 2.30 *Table One*

x
1

Output 2.31 *Table Two*

x	c
2	two

This example code specifies a subquery in the SELECT statement. The subquery specifies to display only values from table All1 that exist in table One. Table All1 is not modified. The query affects outputted rows only:

```
select * from all1 where x in (select x from one);
```

Output 2.32 *Output of the SELECT Subquery*

x	c
1	one

The following example code specifies a subquery in the INSERT statement:

```
insert into two select * from all1 where exists (select * from one);
select * from two;
```

In this example, the content of table Two is expanded to include the contents of table All1 that also exist in table One.

Output 2.33 *Results of the INSERT Subquery*

X	C
2	two
1	one
2	two
3	three
4	four
5	five
6	six

This example code specifies a subquery in the DELETE statement:

```
delete from all1 where x in (select x from one);
select * from all1;
```

The code specifies to delete values from table All1 that match values in column X of table One.

Output 2.34 *Results of the DELETE Subquery*

x	c
2	two
3	three
4	four
5	five
6	six

The following example code specifies a subquery in an UPDATE statement:

```
update all1 set x = (select x from two);
select * from all1;
```

Column X in table All1 is updated with the value of column X in table Two.

Output 2.35 *Results of the UPDATE Subquery*

x	c
1	two
1	three
1	four
1	five
1	six

A non-correlated subquery allows you select, insert, delete, and modify unrelated blocks of data between two or more tables.

FedSQL Value Expressions

Numeric Value Expressions

Numeric value expressions enable you to compute numeric values by using addition (+), subtraction (–), multiplication (*), and division (/) operators. Numeric values can be numeric literals. These values can also be column names, variables, or subqueries as long as the column names, variables, or subqueries evaluate to a numeric value.

The data type of the result of a numeric value expression is based on the data type of the operands.

Here are examples of numeric value expressions.

- –6
- salary * 1.07
- cost + (exp – discount)

Row Value Expressions

A *row value expression*, or *row value constructor*, is one or more value expressions enclosed in parentheses. Multiple value expressions are separated by commas.

A row value constructor can contain the following values.

- *value-expression*
- NULL
- DEFAULT
- *row-subquery*

- (*row-value-constructor*, *row-value-constructor*, …)
- ROW (*row-value-constructor*, *row-value-constructor*, …)

NULL makes the value for the corresponding column in the table null. DEFAULT makes the value for the corresponding column the default value. ARRAY[ ] is valid only if the destination is an array and creates an empty array. The row constructor values other than NULL, DEFAULT, and ARRAY[ ] can be simple values or value expressions.

A row value constructor operates on a list of values or columns rather than a single value or column. You can operate on an entire row at a time or a subset of a row.

Here is an example where you can use row value constructors in the INSERT statement to add multiple values to a table.

```
INSERT INTO inventory
   (prodname, qty, price)
   VALUES ('rice', 3849, .37);
```

This example uses row value constructors in the WHERE clause to compare column values.

```
SELECT * FROM inventory
   WHERE (inventory.prodname, inventory.price)
      =
         (competitor.prodname, inventory.price);
```

Dates and Times in FedSQL

Overview of Dates and Times in FedSQL

FedSQL supports the industry standard conventions for dates, times, and datetimes using the DATE, TIME, and TIMESTAMP data types. All third-party data sources that are supported by FedSQL support these data types. You can create SAS data sets, SPD Engine data sets, and SPD Server tables using these data types, but FedSQL converts these data types to a DOUBLE having a SAS date, time, or datetime format.

FedSQL can read and process SAS dates, time, and datetime values only as values with a data type of DOUBLE. FedSQL cannot convert a SAS date, time, or datetime value with a data type of DOUBLE to a value with a data type of DATE, TIME, or TIMESTAMP.

FedSQL date, time, and datetime functions can provide local current time as well as GMT current time.

FedSQL formats enable dates, times, and datetimes to be formatted in various ways.

FedSQL Date, Time, and Datetime Constants

You write FedSQL date, time, or datetime constants using the following syntax:

DATE '*yyyy-mm-dd*'

TIME '*hh*:*nn*:*ss*[.*fraction*]'

TIMESTAMP ' *yyyy-mm-dd hh*:*mm*:*ss*[.*fraction*]'

where

- *yyyy* is a four-digit year
- *mm* is a two-digit month, 01–12
- *dd* is a two-digit day, 01–31
- *hh* is a two-digit military hour, 00–23
- *nn* is a two-digit minute, 00–59
- *ss* is a two-digit second, 00–59
- *fraction* can be one to ten digits, 0–9, is optional, and represents a fraction of a second

The string portion of the value after the DATE, TIME, or TIMESTAMP keyword must be enclosed in single quotation marks.

In the date constant, the hyphens are required and the length of the date string must be at least 8. You are not required to precede single-digit dates with a zero (0).

In the time constant, the colons are required. If the fraction of a second is not present, the time string must be eight characters long and it can include or exclude the period. If the fraction of second is present, the fraction can be up to nine digits long. The time constant can be between 8 and 18 characters long.

In the timestamp constant, the hyphens in the date are required as well as the colons in the time. If the fraction of a second is not present, it can include or exclude the period. If fraction of a second is present, the fraction can be up to nine digits long. The timestamp constant can be between 19 and 29 characters long.

Here are examples of FedSQL date, time, and timestamp constants:

```
date'2008-01-31'
date'2008-1-1'
time'20:44:59'
timestamp'2007-02-02 07:00:00.7569'
```

The keyword DATETIME can be used as an alias for TIMESTAMP in the timestamp constant. Here is an example: `datetime'2007-02-02 07:00:00.7569'`.

Here is an example of creating a table that includes datetime values:

```
create table bikerace (name char(30), entry_number int,
   registration_date timestamp);
   insert into bikerace values ('Andersen, Mark', 342,
      timestamp'2007-03-15 12:27:33');
   insert into bikerace values ('Steinbek, Mark', 244,
      timestamp'2006-11-27 13:26:19');
```

FedSQL does not return an error message when the year value in a date or timestamp constant is longer than four digits. For example, inserting the value `timestamp'20211-01-01 12:34:56'` does not return an error message. When an overlong year value is inserted, the resulting output value can vary. The results can also be unpredictable and incorrect. Therefore, verify that year values contain only four digits before inserting data.

Using Dates and Times in SAS Data Sets, SPD Engine Data Sets, and SPD Server Tables

When you create a table using the BASE driver or the SPD drivers, you specify columns for date, time, and datetime values by using the DATE, TIME, and TIMESTAMP data types. FedSQL converts values with these data types to a data type of DOUBLE having these SAS formats:

Table 2.15 *SAS Formats Assigned to Date and Time Values in SAS Data Sets, SPD Engine Data Sets, and SPD Server Tables*

FedSQL Date/Time Data Type	SAS Format
DATE	DATE9.
TIME	TIME8.
TIMESTAMP	DATETIME19.2

These SAS formats cannot be altered by using the FedSQL language ALTER TABLE statement. They can be altered only by using protocols in Base SAS to alter formats that are assigned to a column in a SAS data set, SPD Engine data set, or SPD Server table.

The following example creates a SAS data set that contains a date, time, and datetime value using the DATE, TIME, and TIMESTAMP data types, and illustrates how they are displayed using SAS formats:

```
create table basedt (d date, t time, ts timestamp);
insert into basedt values (date '2013-03-14', time '10:31:22',
   timestamp '2013-03-14 13:30:33.222');
select * from basedt;
```

Here is the output:

Output 2.36 *Results for Date, Time, and Timestamp*

The SAS System

D	T	TS
14MAR2013	10:31:22	14AUG13:13:30:33.22

Any format that you might specify for these date, time, and datetime values overrides the format that is stored with these values. The following SELECT statement uses the PUT function to format the datetime value using the DATETIME21.2 format in order to display the four-character year 2008:

```
select put(ts,datetime21.2) from basedt;
```

Here is the output:

Output 2.37 *Results of Using the PUT Statement*

The SAS System

column
14MAR2013:13:30:33.22

Date, Time, and Datetime Functions

Here is a list and description of FedSQL date, time, and datetime functions:

Table 2.16 *Date, Time, and Datetime Functions*

Function	Description
CURRENT_DATE	returns the current date for your time zone.
CURRENT_TIME	returns the current time for your timezone.
CURRENT_TIME_GMT()	returns the current GMT time.
CURRENT_TIMESTAMP	returns the current date and time for your timezone.
CURRENT_TIMESTAMP_GMT()	returns the current GMT date and time.

Function	Description
DAY	returns the numeric day of the month from a date or datetime value.
HOUR	returns the hour from a time or datetime value.
MINUTE	returns the minute from a time or datetime value.
MONTH	returns the numeric month from a date or datetime value.
SECOND	returns the second from a time or datetime value.
YEAR	returns the year from a date or datetime value.

Date, Time, and Datetime Formats for SAS Data Sets, SPD Engine Data Sets, and SPD Server Tables

The following date, time, and datetime formats can be used to format SAS data sets, SPD Engine data sets, and SPD Server tables:

Table 2.17 *Date, Time, and Datetime Functions Arranged by Type*

Type	Language Element	Input	Formatted Result
Date formats	DATE.	date'2013-03-17'	17MAR13
	DATE9.	date'2013-03-17'	17MAR2013
	DATEAMPM21.	timestamp'2013-03-17 14:22:21'	17MAR13:02:22:21 PM
	DATEAMPM23.	timestamp'2013-03-17 14:22:21'	17MAR2013:02:22:21 PM
	DAY.	date'2013-03-17'	17
	DDMMYY.	date'2013-03-17'	17/03/13

Type	Language Element	Input	Formatted Result
	DDMMYB.	date'2013-03-17'	17 03 13
	DDMMYYC.	date'2013-03-17'	17:03:13
	DDMMYYD10.	date'2013-03-17'	17-03-2013
	DDMMYYN.	date'2013-03-17'	17032013
	DDMMYYP.	date'2013-03-17'	17.03.13
	DDMMYYS.	date'2013-03-17'	17/03/13
	DOWNAME.	date'2013-03-17'	Monday
	JULIAN.	date'2013-03-17'	13077
	MMDDYY.	date'2013-03-17'	03/17/13
	MMDDYY10.	date'2013-03-17'	03/17/2013
	MMDDYYB.	date'2013-03-17'	03 17 13
	MMDDYC.	date'2013-03-17'	03:17:13
	MMDDYD.	date'2013-03-17'	03-17-13
	MMDDYN.	date'2013-03-17'	03172013
	MMDDYYP.	date'2013-03-17'	03.17.13
	MMDDYYS.	date'2013-03-17'	03/17/13
	MMYY.	date'2013-03-17'	03M2013
	MMYYC.	date'2013-03-17'	03:2013
	MMYYD.	date'2013-03-17'	03-2013
	MMYYN.	date'2013-03-17'	032013
	MMYYP.	date'2013-03-17'	03.2013
	MMYYS.	date'2013-03-17'	03/2013
	MONNAME.	date'2013-03-17'	March
	MONTH.	date'2013-03-17'	3

Type	Language Element	Input	Formatted Result
	MONYY.	date'2013-03-17'	MAR13
	WEEKDATE.	date'2013-03-17'	Monday, March 17, 2013
	WEEKDATX.	date'2013-03-17'	Monday 17 March 2013
	WEEKDAY.	date'2013-03-17'	2
Datetime formats	DTDATE.	timestamp'2013-03-18 14:22:21'	17MAR13
	DTMONYY.	timestamp'2013-03-18 14:22:21'	MAR13
	DTWKDATX.	timestamp'2013-03-18 14:22:21'	Monday, March 17 2013
	DTYEAR.	timestamp'2013-03-18 14:22:21'	2013
	DTYYQC.	timestamp'2013-03-18 14:22:21'	13:1
Quarter formats	QTR.	date'2013-03-17'	1
	QTRR.	date'2013-03-17'	I
Time formats	HHMM.	time'14:22:21'	14:22
	HOUR.	time'14:22:21'	14
	TIME.	time'14:22:21'	14:22:21
	TIMEAMPM.	time'14:22:21'	2:22:21 PM
	TOD.	time'14:22:21	14:22:21
Year formats	YEAR.	date'2013-03-17'	2013
	YYMM.	date'2013-03-17'	2013M03
	YYMMC.	date'2013-03-17'	2013:03
	YYMMD.	date'2013-03-17'	2013-03

Type	Language Element	Input	Formatted Result
	YYMMN.	date'2013-03-17'	201303
	YYMMP.	date'2013-03-17'	2013.03
	YYMMS.	date'2013-03-17'	2013/03
	YYMMDD.	date'2013-03-17'	13-03-17
	YYMMDDB.	date'2013-03-17'	13 03 17
	YYMMDDC.	date'2013-03-17'	13:03:17
	YYMMDDD.	date'2013-03-17'	13-03-17
	YYMMDDN.	date'2013-03-17'	20130317
	YYMMDDP.	date'2013-03-17'	13.03.17
	YYMMDDS.	date'2013-03-17'	13/03/17
	YYMON.	date'2013-03-17'	2013MAR
Year/Quarter formats	YYQ.	date'2013-03-17'	2013Q1
	YYQC.	date'2013-03-17'	2013:1
	YYQD.	date'2013-03-17'	2013-1
	YYQN.	date'2013-03-17'	20131
	YYQP.	date'2013-03-17'	2013.1
	YYQS.	date'2013-03-17'	2013/1
	YYQR.	date'2013-03-17'	2013QI
	YYQRC.	date'2013-03-17'	2013:I
	YYQRD.	date'2013-03-17'	2013-I
	YYQRN.	date'2013-03-17'	2013I
	YYQRP.	date'2013-03-17'	2013.I
	YYQRS.	date'2013-03-17'	2013/I

DICTIONARY Tables

FedSQL DICTIONARY tables are similar to but different from Base SAS DICTIONARY tables. A FedSQL DICTIONARY table is a Read-only table that contains information about columns, tables, and catalogs, and statistics about tables and their associated indexes.

The following table describes the DICTIONARY tables that are available. For a complete description of all the columns in the DICTIONARY tables, see Appendix 5, “DICTIONARY Table Descriptions,” on page 1187.

Table 2.18 *DICTIONARY Tables*

DICTIONARY table	Description
CATALOGS	contains information about catalogs in the current connection.
COLUMNS	contains information about columns in all known tables.
COLUMN_STATISTICS	contains table and index statistics that are based on the columns in the tables.
STATISTICS	contains table and index statistics that are associated with the tables.
TABLES	contains information about table types and a list of table, catalog, or schema names of tables in the data source.

You can query DICTIONARY tables the same way you query any other table. For all DICTIONARY tables except CATALOG, you can include subsetting with a WHERE clause and ordering the results. Identifiers should be quoted. The following queries are examples.

> **TIP** FedSQL sends table and column names to the underlying databases as uppercase when the table is created, unless you quote the names in the CREATE TABLE statement or the data source supports only lowercase storage. DICTIONARY queries are case-sensitive. Specify table and column names as uppercase unless you know them to be otherwise.

```
select * from dictionary.catalogs where catalog='SALECAT';

select * from dictionary.tables;

select * from dictionary.columns where table_name='YEAREND';

select * from dictionary.columns where table_name='view-name';
```

```
select distinct table_cat, table_schem from dictionary.tables;
```

Note: The DICTIONARY tables rely on data source support for their information. If the data source does not support the underlying function call, no rows are returned. For example, some data sources do not return information for the STATISTICS and COLUMN_STATISTICS dictionary tables.

DICTIONARY queries that do not specify a WHERE clause can take a long time to process, especially if there are a large number of tables in the data source.

Google BigQuery users: Column and table queries that are not qualified with a catalog and a schema issue a separate query for each schema available through the connection. For the best results, qualify your dictionary queries with the catalog and schema as follows:

```
select * from dictionary.columns where table_cat='catalog-name'
and table_name='table-name';
```

FedSQL Implicit Pass-Through Facility

Overview

Implicit pass-through (IP) is the process of translating SQL query code into equivalent data source-specific SQL code so that it can be passed directly to the data source for processing. IP improves query response time and enhances security.

The performance benefits that are provided by IP can be divided into two primary categories: data transfer volume reduction and leveraging of data source-specific capabilities. The volume of data being transferred is reduced by performing the query on the data source. The number of rows that are transferred from the data source to FedSQL can be significantly reduced, thereby decreasing the overall query processing time. The leveraging of data source-specific capabilities, such as massively parallel processing, are specific to a data source. Other examples of special capabilities are advanced join techniques, data partitioning, table statistics, and column statistics. These capabilities often allow the data source to perform the SQL query more quickly than FedSQL.

The security benefit of IP is that every part of an IP query that can be processed is processed on the data source side. This eliminates the need to have its associated tables, which might contain sensitive information, transmitted over to the FedSQL side for query processing.

How to Use FedSQL Implicit Pass-Through

FedSQL IP is performed automatically. You are not required to specify any options to use IP.

Single Source FedSQL Implicit Pass-Through

When a query is accessing a single data source, either the full query is implicitly passed down to the data source or the predicates (for example, the WHERE clauses) are passed down to subset the rows that must be transported into FedSQL for local processing. FedSQL can pass queries implicitly only when the SQL syntax is ANSI-compliant. The following limitations might prevent IP:

- functions that are FedSQL-specific, such as PUT.
- certain aggregate statistics such as SKEWNESS, STUDENTS_T, NMISS, KURTOSIS, CSS, USS, and PROBT.
- mathematical functions such as SIN, COS, ATAN, and TAN.
- ANSI-compliant FedSQL syntax might prevent IP if the data source is not ANSI compliant in that area.

Database tables must be in the same catalog in order to be merged and joined using implicit pass-through. Merges and joins of database tables that exist in different catalogs are processed on the client. A query that contains references to multiple schemas (but only one catalog) can be passed to the data source.

Multiple Source FedSQL Implicit Pass-Through

FedSQL can perform IP on queries that include multiple data sources. This is accomplished by breaking the query into multiple queries and passing these individual queries to their respective drivers. In addition to the restrictions listed in “Single Source FedSQL Implicit Pass-Through ” on page 68, multiple source IP has the following additional limitations:

- A maximum of ten tables can be in one comma join.
- Each side of a set operation (for example, UNION, INTERSECT, and EXCEPT) must have tables from multiple sources for multiple source IP to perform correctly.
- If the query contains a correlated subquery, no multiple source IP is attempted.

FedSQL Explicit Pass-Through Facility

Overview

The FedSQL explicit pass-through facility enables you to connect to a data source and send SQL statements directly to that data source for execution. This facility also enables you to use the syntax of your data source, and it supports any non-ANSI standard SQL that is supported by your data source.

You can create a connection to the language processor. This connection supports standard SQL syntax. This connection can also accept native SQL syntax through the use of the CONNECTION TO component of the SELECT statement's FROM clause, and the EXECUTE statement. Note that on SAS Federation Server, use of these statements is allowed only when the DSN is configured to use personal logins because of security restrictions.

CONNECTION TO Component of the FROM Clause

You should use the CONNECTION TO component of the SELECT statement's FROM clause to submit native SQL requests that produce a result set.

The CONNECTION TO component has the following syntax:

FROM CONNECTION TO *catalog* (*native-syntax*) [[AS] *alias*]

Arguments

catalog	specifies the name of a catalog in the existing FedSQL connection.
native-syntax	specifies a select-type query (not DDL) to be run on the catalog's driver.
alias	provides a name for the result set produced by the native query.

Example:

```
select oo.i, oo.rank, ff.onoff
    from connection to catalog_a
        ( select i, rank() over (order by j) rank from table_a ) oo,
    connection to catalog_b
        ( select distinct i, iif(k > 0.5, 1, 0) as onoff from
table_a ) ff
    where oo.i = ff.i
order by 1;
```

For more information, see the "SELECT Statement" on page 952.

EXECUTE Statement

You should use the EXECUTE statement if the native SQL does not produce a result set, such as DML and DDL statements. In addition, the EXECUTE statement accepts native SQL that produces a result set. For more information, see "EXECUTE Statement" on page 945.

Transactions in FedSQL

Some applications have a requirement to treat groups of updates to a particular data source as a single unit. That is, either the entire group of updates is applied to the data or none of them is applied. Applications can test for errors and execute specific commit and rollback operations to provide that functionality.

FedSQL supports both COMMIT and ROLLBACK statements, which provide data protection by ensuring that updates are either fully applied or rolled back to the pre-transaction state when an operation is interrupted.

A *transaction* is an atomic unit of work. That is, a transaction either completely succeeds or has no effect. After a logical, consistent set of changes has occurred, a transaction is ended either by committing the changes, which makes them permanent, or by canceling the changes, which returns the values that are changed by a transaction to their original state.

When a connection to a database is established, autocommit functionality is the default. That is, each individual FedSQL statement is treated as a transaction. As soon as the statement is executed, if no return code is detected, the transaction is automatically committed. If update problems are detected, FedSQL initiates a rollback of the transaction.

To allow a transaction to be made up of multiple FedSQL statements, the application must turn off autocommit functionality. When autocommit functionality is on, COMMIT and ROLLBACK statements have no effect.

Not all data sources provide transaction management. For example, transaction management is not available for SAS data sets, SPD Engine data sets, and SPD Server tables. However, transaction support is available for data sources such as Oracle and Teradata. See the server administration documentation for information about how to turn off autocommit functionality. For example, see *SAS Federation Server: Administrator's Guide* or the FEDSQL procedure documentation for information about connection options that turn autocommit functionality off.

FedSQL Reserved Words

The following words are reserved as FedSQL language keywords and cannot be used as variable names or in any other way.

Note: You can use delimited identifiers for terms that might otherwise be a reserved word. For example, to use the term “char” other than for a character declaration, you would use it as the delimited identifier “char”. For more information, see “Delimited Identifiers” on page 19.

Table 2.19 *FedSQL Reserved Words A - D*

A	BINARY	COMMENT	**D**
ABORT	BIT	COMMIT	DATABASE
ABSOLUTE	BLOB	COMMITTED	DAY
ACCESS	BOOLEAN	CONDITION	DEALLOCATE
ACTION	BOTH	CONNECT	DEC
ADD	BY	CONSTRAINT	DECIMAL
AFTER	**C**	CONSTRAINTS	DECLARE
AGGREGATE	CACHE	CONVERSION	DEFAULT
ALL	CALL	CONVERT	DEFAULTS
ALLOCATE	CALLED	COPY	DEFERRABLE
ALTER	CARDINALITY	CORR	DEFERRED
ANALYSE	CASCADE	CORRESPONDING	DEFINER
ANALYZE	CASCADED	COVAR_POP	DELETE
AND	CASE	COVAR_SAMP	DELIMITER
ANY	CAST	CREATE	DELIMITERS
ARE	CHAIN	CREATEDB	DENSE_RANK
ARRAY	CHAR	CREATEUSER	DEREF
AS	CHAR_LENGTH	CROSS	DESC
ASC	CHARACTER	CUBE	DESCRIBE
ASENSITIVE	CHARACTER_LENGTH	CUME_DIST	DETERMINISTIC
ASSERTION	CHARACTERISTICS	CURRENT	DISCONNECT
ASSIGNMENT	CHECK	CURRENT_DATE	DISTINCT
ASYMMETRIC	CHECKPOINT	CURRENT_DEFAULT_TRANSFORM_GROUP	DO
AT	CLASS	CURRENT_PATH	DOMAIN
ATOMIC	CLOB	CURRENT_ROLE	DOUBLE
AUTORIZATION			DROP

B	CLOSE	CURRENT_TIME	DYNAMIC
BACKWARD	CLUSTER	CURRENT_TIMESTAMP	
BEFORE	COALESCE	CURRENT_TRANSFORM_GROUP_FOR_TYPE	
BEGIN	COLLATE	CURRENT_USER	
BETWEEN	COLLECT	CURSOR	
BIGINT	COLUMN	CYCLE	

Table 2.20 FedSQL Reserved Words E - O

E	**G**	**J**	**N**
EACH	GET	JOIN	NAMES
ELEMENT	GLOBAL	**K**	NATIONAL
ELSE	GRANT	KEY	NATURAL
ENCODING	GROUP	**L**	NCHAR
ENCRYPTED	GROUPING	LANCOMPILER	NCLOB
END	**H**	LANGUAGE	NEW
END-EXEC	HANDLER	LARGE	NEXT
ESCAPE	HAVING	LAST	NO
EVERY	HOLD	LATERAL	NOCREATEDB
EXCEPT	**I**	LEADING	NOCREATEUSER
EXCLUDING	ILIKE	LEFT	NONE
EXCLUSIVE	IMMEDIATE	LEVEL	NORMALIZE
EXEC	IMMUTABLE	LIKE	NOT
EXECUTE	IMPLICIT	LIMIT	NOTHING
EXISTS	IN	LISTEN	NOTIFY
EXPLAIN	INCLUDING	LOAD	NOTNULL
EXTERNAL	INCREMENT	LOCAL	NULL
EXTRACT	INDEX	LOCALTIME	NULLIF
F	INDICATOR	LOCALTIMESTAMP	NUMERIC
FALSE	INHERITS	LOCATION	**O**
FETCH	INITIALLY	LOCK	OF
FILTER	INNER	**M**	OFF
FIRST	INOUT	MATCH	OFFSET
FLOAT	INPUT	MAXVALUE	OIDS
FOR	INSENSITIVE	MEMBER	OLD
FORCE	INSERT	MERGE	ON
FOREIGN	INSTEAD	METHOD	ONLY
FORWARD	INT		OPEN

FREE	INTEGER	MINUTE	OPERATOR
FREEZE	INTERSECT	MINVALUE	OPTION
FROM	INTERSECTION	MODE	OR
FULL	INTERVAL	MODIFIES	ORDER
FUNCTION	INTO	MODULE	OUT
FUSION	INVOKER	MONTH	OUTER
	IS	MOVE	OVER
	ISNULL	MULTISET	OVERLAPS
	ISOLATION		OVERLAY
			OWNER

Table 2.21 *FedSQL Reserved Words P - Z*

P	REPLACE	STATEMENT	UNION
PARAMETER	RESET	STATIC	UNIQUE
PARTIAL	RESTART	STATISTICS	UNKNOWN
PARTITION	RESTRICT	STDDEV_POP	UNLISTEN
PASSWORD	RESULT	STDDEV_SAMP	UNNEST
PATH	RETURN	STDIN	UNTIL
PENDANT	RETURNS	STDOUT	UPDATE
PERCENT_RANK	REVOKE	STORAGE	USAGE
PERCENTILE_CONT	RIGHT	STRICT	USER
PERCENTILE_DESC	ROLLBACK	SUBMULTISET	USING
PLACING	ROLLUP	SUBSTRING	**V**
POSITION	ROW	SYMMETRIC	VACUUM
PRECISION	ROWS	SYSID	VALID
PREPARE	ROW_NUMBER	SYSTEM	VALIDATOR
PRESERVE	RULE	SYSTEM_USER	VALUE
PRIMARY	**S**	**T**	VALUES
PRIOR	SCHEMA	TABLE	VARCHAR
PRIVILEGES	SCOPE	TABLESAMPLE	VARYING
PROCEDURAL	SCROLL	TEMP	VAR_POP
PROCEDURE	SEARCH	TEMPLATE	VAR_SAMP
R	SECOND	TEMPORARY	VERBOSE
RANK	SECURITY	THEN	VERSION
READ	SELECT	TIME	VIEW
READS	SENSITIVE	TIMESTAMP	VOLATILE
REAL	SPECIFIC	TIMEZONE_HOUR	**W**
RECHECK	SPECIFICTYPE	TIMEZONE_MINUTE	WHEN
RECURSIVE	SEQUENCE	TO	WHENEVER

REF
REFERENCES
REFERENCING
REGR_AVGX
REGR_AVGY
REGR_COUNT
REGR_INTERCEPT
REGR_R2
REGR_SLOPE
REGR_SXX
REGR_SXY
REGR_SYY
REINDEX
RELATIVE
RENAME
SERIALIZABLE
SESSION
SESSION_USER
SET
SETOF
SHARE
SHOW
SIMILAR
SIMPLE
SMALLINT
SOME
SQLEXCEPTION
SQLSTATE
SQLWARNING
STABLE
START
TOAST
TRAILING
TRANSACTION
TRANSLATE
TRANSLATION
TREAT
TRIGGER
TRIM
TRUE
TRUNCATE
TRUSTED
TYPE
U
UESCAPE
UNENCRYPTED
WHERE
WIDTH_BUCKET
WINDOW
WITH
WITHIN
WITHOUT
WORK
WRITE
Y
YEAR
Z
ZONE

3

FedSQL Formats

Overview of Formats

A format is an instruction that FedSQL uses to write data values. You use formats to control the written appearance of data values, or, in some cases, to group data values together for analysis. For example, the ROMANw. format, which converts numeric values to roman numerals, writes the numeric value 2013 as `MMXIII`.

When you create a SAS data set, SPD Engine data set, or SPD Server table with FedSQL, formatting instructions can be stored with the data set, and automatically applied by the SAS Output Delivery System (ODS) when the data set is displayed in a Base SAS session. Formats must be explicitly specified in the PUT function for the other SAS/ACCESS engines, because the data source drivers do not store information about formats and informats.

General Format Syntax

FedSQL formats have the following syntax:

[$] *format* [*w*] . [*d*]

Arguments

$
indicates a character format; its absence indicates a numeric informat.

format
names the format. The format is a SAS format, a FedSQL format, or a user-defined format that was previously defined with the INVALUE statement in PROC FORMAT. For more information about user-defined formats, see PROC FORMAT in *Base SAS Procedures Guide*.

w
specifies the format width, which for most formats is the number of columns in the input data.

d
specifies a decimal scaling factor in the numeric formats. FedSQL divides the input data by 10 to the power of *d*.

Tip When the value of *d* is greater than 15, the precision of the decimal value after the 15th decimal place might not be accurate.

Formats always contain a period (.) as a part of the name. If you omit the *w* and the *d* values from the format, SAS uses default values. The *d* value that you specify with a format tells FedSQL to display that many decimal places, regardless of how many

decimal places are in the data. Formats never change or truncate the internally stored data values.

For example, in DOLLAR10.2, the *w* value of 10 specifies a maximum of 10 columns for the value. The *d* value of 2 specifies that two of these columns are for the decimal part of the value, which leaves eight columns for all the remaining characters in the value. This includes the decimal point, the remaining numeric value, a minus sign if the value is negative, the dollar sign, and commas, if any.

If the format width is too narrow to represent a value, FedSQL tries to squeeze the value into the space available. Character formats truncate values on the right. Numeric formats sometimes revert to the BESTw. format. The BESTw. format is the default format for writing numeric values. BESTw. rounds the value, and if SAS can display at least one significant digit in the decimal portion within the width specified, BESTw. produces the result in decimal. Otherwise, it produces the result in scientific notation. SAS always stores the complete value regardless of the format that you use to represent it. At least 3 columns must be available for the BESTw. format to be applied. FedSQL prints blanks if you do not specify an adequate width. To illustrate, the following request:

```
select put(12345, 3.);
```

returns the output **1E4**. Meanwhile, this request:

```
select put(12345, 2.);
```

returns a blank value.

If you use an incompatible format, such as using a numeric format to write character values, FedSQL first attempts to use an analogous format of the other type. If this is not feasible, an error message that describes the problem appears in the SAS log.

Using Formats in FedSQL

How to Store, Change, Delete, and Use Stored Formats

Storage of format metadata is supported in SAS data sets, SPD Engine data sets, and SPD Server tables only. As a result, when they are used in Base SAS, SAS data sets that were created with FedSQL behave the same as data sets that were created with SAS.

You specify formats in the CREATE TABLE statement as an attribute of the HAVING clause. For more information, see “CREATE TABLE Statement” on page 915. For example, in the following statement, the column *profit* is declared with the EURO13.2 format.

```
create table monthly (profit double having format euro13.2);
```

To change or remove a stored format, you must use Base SAS. When you want to display a different format for a column that has a stored format value when reading a table with FedSQL, use the PUT function.

FedSQL supports stored formats as follows:

- Both user-defined formats and formats that are supplied by SAS can be stored.
- All formats that are supplied by SAS can be stored. For a list of formats, see *SAS Formats and Informats: Reference*. FedSQL does not validate the formats. If the stored format is invalid, an error occurs, but only when the invalid format is used in the client application.
- Formats can be stored only for columns of data types CHAR and DOUBLE.
- To access the stored formats, you must have a Base SAS session available. The Base SAS session contains the SAS format definitions and SAS catalog file that stores the user-defined format definitions.
- You can store and retrieve format names. The format name is associated with a column by storing the format as a metadata attribute on the column. The metadata then can be retrieved for subsequent operations.

How to Format Output with the PUT Function

The PUT function enables you to associate a format with data in third-party data sources, as well as with SAS data. Formats are specified as arguments in the PUT function to generate formatted data. In the following example, the PUT function returns the formatted value of `4503945867` using the DOLLAR17.2 format. The example returns the value `$4,503,945,867.00`.

```
select put(4503945867, dollar17.2);
```

FedSQL supports formats that are specified with the PUT function as follows:

- If the PUT function is used without a format, an error occurs.
- The PUT function supports a subset of the formats that are available for Base SAS when the FedSQL language is executed outside a Base SAS session. For a list, see “Formats Supported with the PUT Function, by Category” on page 85.
- Formats can be associated with any of the data types that are supported by FedSQL. However, the data types are converted. Any value that is passed to the PUT function with a numeric format is converted to NVARCHAR, VARBINARY, or BINARY. The type conversions are carried out based on the format name. Any value that is passed with a character format to the PUT function is converted to NVARCHAR.
- The format that is specified in PUT is transient. The PUT function does not affect the stored data.
- The PUT function does not require a Base SAS session to be available; however, the functionality is limited when a session is not available.

See also the “PUT Function” on page 769.

Validation of FedSQL Formats

When a format is stored in a data set using FedSQL, no validation occurs. When metadata for a column is requested, the format name is returned without validation. The format is validated at execution time.

The PUT function validates the specified format upon use.

FedSQL Format Examples

```
create table mybase.sales
   ( prod char(10) having format $10.,
     totals double having format dollar6.
   );

select put (totals, dollar10.) as totals from mybase.sales;

select put(13500, comma6.);
```

Using a User-Defined Format with the PUT Function

You can use the SAS FORMAT procedure to define custom formats that replace raw data values with formatted character values. For example, the following PROC FORMAT code creates a custom numeric format called DEPTNO. that maps department codes to their corresponding department name.

```
proc format;
   value deptno
   10 = 'Sales'
   20 = 'Research'
   30 = 'Accounting'
   40 = 'Operations';
run;
```

The resulting user-defined format can be stored in a SAS data set, SPD Engine data set, or SPD Server table, or it can be applied to a third-party data source by using the PUT function. The following code uses the PUT function and DEPTNO. format to display the numeric department codes in the DEPT column of the EMPLOYEES table as their corresponding character-based department name.

```
select emp_name, hire_date, put(dept, deptno.) as dept
```

```
from employees limit 4;
quit;
```

A few rows of the source EMPLOYEES table is shown in Figure 3.4 on page 81. The output of the PUT function is shown in Figure 3.5 on page 81.

Figure 3.1 *Content of the Source EMPLOYEES Table*

EMP_NAME	HIRE_DATE	DEPT
Jim Barnes	26NOV2004	10
Clifford James	26NOV2004	20
Barbara Sandman	26NOV2004	30
William Baylor	26NOV2004	40

Figure 3.2 *Content of the Employees Table After the PUT Function Is Applied*

EMP_NAME	HIRE_DATE	DEPT
Jim Barnes	26NOV2004	Sales
Clifford James	26NOV2004	Research
Barbara Sandman	26NOV2004	Accounting
William Baylor	26NOV2004	Operations

For more information about how to create your own format in SAS, see PROC FORMAT in *Base SAS Procedures Guide*.

Format Categories

Formats can be categorized by the types of values that they operate on. Each FedSQL format belongs to one of the following categories:

CAS
: identifies formats that can be used on the CAS server.

Character
: writes character data values from character variables.

Date and Time
: writes character data values from character variables.

Numeric
: writes numeric data values from numeric variables.

NLS Formats Supported by FedSQL

National Language Support (NLS) is a set of features that enable a software product to function properly in every global market for which the product is targeted. The SAS System contains NLS features to ensure that SAS applications can be written so that they conform to local language conventions. Typically, software that is written in the English language works well for users who use both the English language and also data that is formatted using the conventions that are observed in the United States. However, without NLS, these products might not work well for users in other regions of the world. NLS in SAS enables regions such as Asia and Europe to process data successfully in their native languages and environments. The FedSQL language supports the following NLS formats. For more information, see *SAS National Language Support (NLS): Reference Guide*.

Category	Language Element	Description
Date and Time	NLDATE*w*.	Converts a SAS date value to the date value of the specified locale, and then writes the date value as a date.
	NLDATEMD*w*.	Converts the SAS date value to the date value of the specified locale, and then writes the value as the name of the month and the day of the month.
	NLDATEMN*w*.	Converts a SAS date value to the date value of the specified locale, and then writes the value as the name of the month.
	NLDATEW*w*.	Converts a SAS date value to the date value of the specified locale, and then writes the value as the date and the day of the week.
	NLDATEWN*w*.	Converts the SAS date value to the date value of the specified locale, and then writes the date value as the day of the week.
	NLDATEYM*w*.	Converts the SAS date value to the date value of the specified locale, and then writes the date value as the year and the name of the month.
	NLDATEYQ*w*.	Converts the SAS date value to the date value of the specified locale, and then writes the date value as the year and the quarter.

Category	Language Element	Description
	NLDATEYR*w*.	Converts the SAS date value to the date value of the specified locale, and then writes the date value as the year.
	NLDATEYW*w*.	Converts the SAS date value to the date value of the specified locale, and then writes the date value as the year and the week.
	NLDATMAP*w*.	Converts a SAS datetime value to the datetime value of the specified locale, and then writes the value as a datetime with a.m. or p.m.
	NLDATMDT*w*.	Converts the SAS datetime value to the datetime value of the specified locale, and then writes the value as the name of the month, day of the month, and year.
	NLDATMMD*w*.	Converts the SAS datetime value to the datetime value of the specified locale, and then writes the value as the name of the month and the day of the month.
	NLDATMMN*w*.	Converts the SAS datetime value to the datetime value of the specified locale, and then writes the value as the name of the month.
	NLDATMTM*w*.	Converts the time portion of a SAS datetime value to the time-of-day value of the specified locale, and then writes the value as a time of day.
	NLDATM*w*.	Converts a SAS datetime value to the datetime value of the specified locale, and then writes the value as a datetime.
	NLDATMW*w*.	Converts SAS datetime values to the locale sensitive datetime string as the day of the week and the datetime.
	NLDATMWN*w*.	Converts a SAS datetime value to the datetime value of the specified locale, and then writes the value as the day of the week.
	NLDATMYM*w*.	Converts the SAS datetime value to the datetime value of the specified locale, and then writes the value as the year and the name of the month.
	NLDATMYQ*w*.	Converts the SAS datetime value to the datetime value of the specified locale, and

Category	Language Element	Description
		then writes the value as the year and the quarter of the year.
	NLDATMYR*w*.	Converts the SAS datetime value to the datetime value of the specified locale, and then writes the value as the year.
	NLDATMYW*w*.	Converts the SAS datetime value to the datetime value of the specified locale, and then writes the value as the year and the name of the week.
	NLTIMAP*w*.	Converts a SAS time value to the time value of a specified locale, and then writes the value as a time value with a.m. or p.m. NLTIMAP also converts SAS date-time values.
	NLTIME*w*.	Converts a SAS time value to the time value of the specified locale, and then writes the value as a time value. NLTIME also converts SAS date-time values.
Numeric	NLBEST*w*.	Writes the best numerical notation based on the locale.
	NLMNY*w.d*	Writes the monetary format of the local expression in the specified locale using local currency.
	NLMNYI*w.d*	Writes the monetary format of the international expression in the specified locale.
	NLNUM*w.d*	Writes the numeric format of the local expression in the specified locale.
	NLNUMI*w.d*	Writes the numeric format of the international expression in the specified locale.
	NLPCT*w.d*	Writes percentage data of the local expression in the specified locale.
	NLPCTI*w.d*	Writes percentage data of the international expression in the specified locale.
	NLPCTN*w.d*	Produces percentages, using a minus sign for negative values.

Category	Language Element	Description
	NLPCTP*w.d*	Writes locale-specific numeric values as percentages. Writes locale-specific numeric values as percentages.
	NLPVALUE*w.d*	Writes p-values of the local expression in the specified locale.
	NLSTRMON*w.d*	Writes the month name in the specified locale.
	NLSTRQTR*w.d*	Writes a numeric value as the quarter-of-the-year in the specified locale.
	NLSTRWK*w.d*	Writes a numeric value as the day-of-the-week in the specified locale.

Formats Supported with the PUT Function, by Category

Category	Language Elements	Description
CAS	$BASE64Xw. Format (p. 94)	Converts character data into ASCII text by using Base 64 encoding.
	$BINARYw. Format (p. 96)	Converts character data to binary representation.
	$CHARw. Format (p. 97)	Writes standard character data.
	$HEXw. Format (p. 98)	Converts character data to hexadecimal representation.
	$OCTALw. Format (p. 99)	Converts character data to octal representation.
	$QUOTEw. Format (p. 101)	Writes data values that are enclosed in single quotation marks.
	$REVERJw. Format (p. 102)	Writes character data in reverse order and preserves blanks.
	$REVERSw. Format (p. 103)	Writes character data in reverse order and left aligns.
	$UPCASEw. Format (p. 104)	Converts character data to uppercase.
	$w. Format (p. 105)	Writes standard character data.

Category	Language Elements	Description
	BESTw. Format (p. 106)	SAS chooses the best notation.
	BESTDw.p Format (p. 108)	Prints numeric values, lining up decimal places for values of similar magnitude, and prints integers without decimals.
	BINARYw. Format (p. 110)	Converts numeric values to binary representation.
	COMMAw.d Format (p. 111)	Writes numeric values with a comma that separates every three digits and a period that separates the decimal fraction.
	COMMAXw.d Format (p. 112)	Writes numeric values with a period that separates every three digits and a comma that separates the decimal fraction.
	Dw.p Format (p. 114)	Prints variables, possibly with a great range of values, lining up decimal places for values of similar magnitude.
	DATEw. Format (p. 115)	Writes SAS date values in the form ddmmmyy, ddmmmyyyy, or dd-mmm-yyyy.
	DATEAMPMw.d Format (p. 117)	Writes SAS datetime values in the form ddmmmyy:hh:mm:ss.ss with AM or PM.
	DATETIMEw.d Format (p. 119)	Writes SAS datetime values in the form ddmmmyy:hh:mm:ss.ss.
	DAYw. Format (p. 121)	Writes SAS date values as the day of the month.
	DDMMYYw. Format (p. 122)	Writes SAS date values in the form ddmm[yy]yy or dd/mm/[yy]yy, where a forward slash is the separator and the year appears as either 2 or 4 digits.
	DDMMYYxw. Format (p. 124)	Writes SAS date values in the form ddmm[yy]yy or ddXmmX[yy]yy, where X represents a specified separator and the year appears as either 2 or 4 digits.
	DOLLARw.d Format (p. 126)	Writes numeric values with a leading dollar sign, a comma that separates every three digits, and a period that separates the decimal fraction.
	DOLLARXw.d Format (p. 127)	Writes numeric values with a leading dollar sign, a period that separates every three digits, and a comma that separates the decimal fraction.
	DOWNAMEw. Format (p. 129)	Writes SAS date values as the name of the day of the week.
	DTDATEw. Format (p. 130)	Expects a SAS datetime value as input and writes the SAS date values in the form ddmmmyy or ddmmmyyyy.
	DTMONYYw. Format (p. 131)	Writes the date part of a SAS datetime value as the month and year in the form mmmyy or mmmyyyy.

Category	Language Elements	Description
	DTWKDATXw. Format (p. 133)	Writes the date part of a SAS datetime value as the day of the week and the date in the form day-of-week, dd month-name yy (or yyyy).
	DTYEARw. Format (p. 134)	Writes the date part of a SAS datetime value as the year in the form yy or yyyy.
	DTYYQCw. Format (p. 136)	Writes the date part of a SAS datetime value as the year and the quarter, and separates them with a colon (:).
	Ew. Format (p. 137)	Writes numeric values in scientific notation.
	EUROw.d Format (p. 138)	Writes numeric values with a leading euro symbol (€), a comma that separates every three digits, and a period that separates the decimal fraction.
	EUROXw.d Format (p. 140)	Writes numeric values with a leading euro symbol (€), a period that separates every three digits, and a comma that separates the decimal fraction.
	FLOATw.d Format (p. 141)	Generates a native single-precision, floating-point value by multiplying a number by 10 raised to the dth power.
	FRACTw. Format (p. 142)	Converts numeric values to fractions.
	HEXw. Format (p. 143)	Converts real binary (floating-point) values to hexadecimal representation.
	HHMMw.d Format (p. 145)	Writes SAS time values as hours and minutes in the form hh:mm.
	HOURw.d Format (p. 147)	Writes SAS time values as hours and decimal fractions of hours.
	IEEEw.d Format (p. 148)	Generates an IEEE floating-point value by multiplying a number by 10 raised to the dth power.
	JULIANw. Format (p. 149)	Writes SAS date values as Julian dates in the form yyddd or yyyyddd.
	MMDDYYw. Format (p. 151)	Writes SAS date values in the form mmdd[yy]yy or mm/dd/[yy]yy, where a forward slash is the separator and the year appears as either 2 or 4 digits.
	MMDDYYxw. Format (p. 152)	Writes SAS date values in the form mmdd[yy]yy or mmXddX[yy]yy, where X represents a specified separator and the year appears as either 2 or 4 digits.
	MMSSw.d Format (p. 154)	Writes SAS time values as the number of minutes and seconds since midnight.
	MMYYw. Format (p. 155)	Writes SAS date values in the form mmM[yy]yy, where M is the separator and the year appears as either 2 or 4 digits.

Category	Language Elements	Description
	MMYYxw. Format (p. 157)	Writes SAS date values in the form mm[yy]yy or mmX[yy]yy. The x in the format name is a character that represents the special character. The special character separates the month and the year. That special character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits.
	MONNAMEw. Format (p. 159)	Writes SAS date values as the name of the month.
	MONTHw. Format (p. 160)	Writes SAS date values as the month of the year.
	MONYYw. Format (p. 161)	Writes SAS date values as the month and the year in the form mmmyy or mmmyyyy.
	NEGPARENw.d Format (p. 162)	Writes negative numeric values in parentheses.
	NENGOw. Format (p. 164)	Writes SAS date values as Japanese dates in the form e.yymmdd.
	OCTALw. Format (p. 165)	Converts numeric values to octal representation.
	PERCENTw.d Format (p. 166)	Writes numeric values as percentages.
	PERCENTNw.d Format (p. 168)	Produces percentages, using a minus sign for negative values.
	QTRw. Format (p. 169)	Writes SAS date values as the quarter of the year.
	QTRRw. Format (p. 170)	Writes SAS date values as the quarter of the year in Roman numerals.
	ROMANw. Format (p. 171)	Writes numeric values as roman numerals.
	SIZEKw.d Format (p. 172)	Writes a numeric value in the form nK for kilobytes.
	TIMEw.d Format (p. 173)	Writes SAS time values as hours, minutes, and seconds in the form hh:mm:ss.ss using the military 24-hour clock.
	TIMEAMPMw.d Format (p. 175)	Writes SAS time values as hours, minutes, and seconds in the form hh:mm:ss.ss with AM or PM.
	TODw.d Format (p. 177)	Writes SAS time values and the time portion of SAS datetime values in the form hh:mm:ss.ss.
	VAXRBw.d Format (p. 178)	Writes real binary (floating-point) data in VMS format.
	w.d Format (p. 179)	Writes standard numeric data one digit per byte.
	WEEKDATEw. Format (p. 181)	Writes SAS date values as the day of the week and the date in the form day-of-week, month-name dd, yy (or yyyy).

Category	Language Elements	Description
	WEEKDATXw. Format (p. 182)	Writes SAS date values as the day of the week and date in the form day-of-week, dd month-name yy (or yyyy).
	WEEKDAYw. Format (p. 184)	Writes SAS date values as the day of the week.
	YEARw. Format (p. 185)	Writes SAS date values as the year.
	YENw.d Format (p. 186)	Writes numeric values with yen signs, commas, and decimal points.
	YYMMw. Format (p. 187)	Writes SAS date values in the form [yy]yyMmm, where M is the separator and the year appears as either 2 or 4 digits.
	YYMMxw. Format (p. 189)	Writes SAS date values in the form [yy]yymm or [yy]yy-mm. The x in the format name represents the special character that separates the year and the month. This special character can be a hyphen (-), period (.), slash(/), colon(:), or no separator. The year can be either 2 or 4 digits.
	YYMMDDw. Format (p. 191)	Writes SAS date values in the form yymmdd or [yy]yy-mm-dd, where a hyphen (-) is the separator and the year appears as either 2 or 4 digits.
	YYMMDDxw. Format (p. 193)	Writes date values in the form [yy]yymmdd or [yy]yy-mm-dd. The x in the format name is a character that represents the special character which separates the year, month, and day. This special character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits.
	YYMONw. Format (p. 195)	Writes SAS date values in the form yymmm or yyyymmm.
	YYQw. Format (p. 196)	Writes SAS date values in the form [yy]yyQq, where Q is the separator, the year appears as either 2 or 4 digits, and q is the quarter of the year.
	YYQxw. Format (p. 197)	Writes SAS date values in the form [yy]yyq or [yy]yy-q. The x in the format name is a character that represents the special character that separates the year and the quarter of the year. This character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits.
	YYQRw. Format (p. 199)	Writes SAS date values in the form [yy]yyQqr, where Q is the separator, the year appears as either 2 or 4 digits, and qr is the quarter of the year expressed in roman numerals.
	YYQRxw. Format (p. 201)	Writes date values in the form [yy]yyqr or [yy]yy-qr. The x in the format name is a character that represents the special character that separates the year and the quarter of the year. This character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits and qr is the quarter of the year in roman numerals.

Category	Language Elements	Description
	YYQZw. Format (p. 202)	Writes SAS date values in the form [yy] yyqq. The year appears as 2 or 4 digits, and qq is the quarter of the year.
	Zw.d Format (p. 204)	Writes standard numeric data with leading 0s.
Character	$BASE64Xw. Format (p. 94)	Converts character data into ASCII text by using Base 64 encoding.
	$BINARYw. Format (p. 96)	Converts character data to binary representation.
	$CHARw. Format (p. 97)	Writes standard character data.
	$HEXw. Format (p. 98)	Converts character data to hexadecimal representation.
	$OCTALw. Format (p. 99)	Converts character data to octal representation.
	$QUOTEw. Format (p. 101)	Writes data values that are enclosed in single quotation marks.
	$REVERJw. Format (p. 102)	Writes character data in reverse order and preserves blanks.
	$REVERSw. Format (p. 103)	Writes character data in reverse order and left aligns.
	$UPCASEw. Format (p. 104)	Converts character data to uppercase.
	$w. Format (p. 105)	Writes standard character data.
Date and Time	DATEw. Format (p. 115)	Writes SAS date values in the form ddmmmyy, ddmmmyyyy, or dd-mmm-yyyy.
	DATEAMPMw.d Format (p. 117)	Writes SAS datetime values in the form ddmmmyy:hh:mm:ss.ss with AM or PM.
	DATETIMEw.d Format (p. 119)	Writes SAS datetime values in the form ddmmmyy:hh:mm:ss.ss.
	DAYw. Format (p. 121)	Writes SAS date values as the day of the month.
	DDMMYYw. Format (p. 122)	Writes SAS date values in the form ddmm[yy]yy or dd/mm/[yy]yy, where a forward slash is the separator and the year appears as either 2 or 4 digits.
	DDMMYYxw. Format (p. 124)	Writes SAS date values in the form ddmm[yy]yy or ddXmmX[yy]yy, where X represents a specified separator and the year appears as either 2 or 4 digits.
	DOWNAMEw. Format (p. 129)	Writes SAS date values as the name of the day of the week.
	DTDATEw. Format (p. 130)	Expects a SAS datetime value as input and writes the SAS date values in the form ddmmmyy or ddmmmyyyy.

Category	Language Elements	Description
	DTMONYYw. Format (p. 131)	Writes the date part of a SAS datetime value as the month and year in the form mmmyy or mmmyyyy.
	DTWKDATXw. Format (p. 133)	Writes the date part of a SAS datetime value as the day of the week and the date in the form day-of-week, dd month-name yy (or yyyy).
	DTYEARw. Format (p. 134)	Writes the date part of a SAS datetime value as the year in the form yy or yyyy.
	DTYYQCw. Format (p. 136)	Writes the date part of a SAS datetime value as the year and the quarter, and separates them with a colon (:).
	HHMMw.d Format (p. 145)	Writes SAS time values as hours and minutes in the form hh:mm.
	HOURw.d Format (p. 147)	Writes SAS time values as hours and decimal fractions of hours.
	JULIANw. Format (p. 149)	Writes SAS date values as Julian dates in the form yyddd or yyyyddd.
	MMDDYYw. Format (p. 151)	Writes SAS date values in the form mmdd[yy]yy or mm/dd/[yy]yy, where a forward slash is the separator and the year appears as either 2 or 4 digits.
	MMDDYYxw. Format (p. 152)	Writes SAS date values in the form mmdd[yy]yy or mmXddX[yy]yy, where X represents a specified separator and the year appears as either 2 or 4 digits.
	MMSSw.d Format (p. 154)	Writes SAS time values as the number of minutes and seconds since midnight.
	MMYYw. Format (p. 155)	Writes SAS date values in the form mmM[yy]yy, where M is the separator and the year appears as either 2 or 4 digits.
	MMYYxw. Format (p. 157)	Writes SAS date values in the form mm[yy]yy or mmX[yy]yy. The x in the format name is a character that represents the special character. The special character separates the month and the year. That special character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits.
	MONNAMEw. Format (p. 159)	Writes SAS date values as the name of the month.
	MONTHw. Format (p. 160)	Writes SAS date values as the month of the year.
	MONYYw. Format (p. 161)	Writes SAS date values as the month and the year in the form mmmyy or mmmyyyy.
	NENGOw. Format (p. 164)	Writes SAS date values as Japanese dates in the form e.yymmdd.
	QTRw. Format (p. 169)	Writes SAS date values as the quarter of the year.

Category	Language Elements	Description
	QTRRw. Format (p. 170)	Writes SAS date values as the quarter of the year in Roman numerals.
	TIMEw.d Format (p. 173)	Writes SAS time values as hours, minutes, and seconds in the form hh:mm:ss.ss using the military 24-hour clock.
	TIMEAMPMw.d Format (p. 175)	Writes SAS time values as hours, minutes, and seconds in the form hh:mm:ss.ss with AM or PM.
	TODw.d Format (p. 177)	Writes SAS time values and the time portion of SAS datetime values in the form hh:mm:ss.ss.
	WEEKDATEw. Format (p. 181)	Writes SAS date values as the day of the week and the date in the form day-of-week, month-name dd, yy (or yyyy).
	WEEKDATXw. Format (p. 182)	Writes SAS date values as the day of the week and date in the form day-of-week, dd month-name yy (or yyyy).
	WEEKDAYw. Format (p. 184)	Writes SAS date values as the day of the week.
	YEARw. Format (p. 185)	Writes SAS date values as the year.
	YYMMw. Format (p. 187)	Writes SAS date values in the form [yy]yyMmm, where M is the separator and the year appears as either 2 or 4 digits.
	YYMMxw. Format (p. 189)	Writes SAS date values in the form [yy]yymm or [yy]yy-mm. The x in the format name represents the special character that separates the year and the month. This special character can be a hyphen (-), period (.), slash(/), colon(:), or no separator. The year can be either 2 or 4 digits.
	YYMMDDw. Format (p. 191)	Writes SAS date values in the form yymmdd or [yy]yy-mm-dd, where a hyphen (-) is the separator and the year appears as either 2 or 4 digits.
	YYMMDDxw. Format (p. 193)	Writes date values in the form [yy]yymmdd or [yy]yy-mm-dd. The x in the format name is a character that represents the special character which separates the year, month, and day. This special character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits.
	YYMONw. Format (p. 195)	Writes SAS date values in the form yymmm or yyyymmm.
	YYQw. Format (p. 196)	Writes SAS date values in the form [yy]yyQq, where Q is the separator, the year appears as either 2 or 4 digits, and q is the quarter of the year.
	YYQxw. Format (p. 197)	Writes SAS date values in the form [yy]yyq or [yy]yy-q. The x in the format name is a character that represents the special character that separates the year and the quarter of the year. This character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits.

Category	Language Elements	Description
	YYQRw. Format (p. 199)	Writes SAS date values in the form [yy]yyQqr, where Q is the separator, the year appears as either 2 or 4 digits, and qr is the quarter of the year expressed in roman numerals.
	YYQRxw. Format (p. 201)	Writes date values in the form [yy]yyqr or [yy]yy-qr. The x in the format name is a character that represents the special character that separates the year and the quarter of the year. This character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits and qr is the quarter of the year in roman numerals.
	YYQZw. Format (p. 202)	Writes SAS date values in the form [yy] yyqq. The year appears as 2 or 4 digits, and qq is the quarter of the year.
Numeric	BESTw. Format (p. 106)	SAS chooses the best notation.
	BESTDw.p Format (p. 108)	Prints numeric values, lining up decimal places for values of similar magnitude, and prints integers without decimals.
	BINARYw. Format (p. 110)	Converts numeric values to binary representation.
	COMMAw.d Format (p. 111)	Writes numeric values with a comma that separates every three digits and a period that separates the decimal fraction.
	COMMAXw.d Format (p. 112)	Writes numeric values with a period that separates every three digits and a comma that separates the decimal fraction.
	Dw.p Format (p. 114)	Prints variables, possibly with a great range of values, lining up decimal places for values of similar magnitude.
	DOLLARw.d Format (p. 126)	Writes numeric values with a leading dollar sign, a comma that separates every three digits, and a period that separates the decimal fraction.
	DOLLARXw.d Format (p. 127)	Writes numeric values with a leading dollar sign, a period that separates every three digits, and a comma that separates the decimal fraction.
	Ew. Format (p. 137)	Writes numeric values in scientific notation.
	EUROw.d Format (p. 138)	Writes numeric values with a leading euro symbol (€), a comma that separates every three digits, and a period that separates the decimal fraction.
	EUROXw.d Format (p. 140)	Writes numeric values with a leading euro symbol (€), a period that separates every three digits, and a comma that separates the decimal fraction.
	FLOATw.d Format (p. 141)	Generates a native single-precision, floating-point value by multiplying a number by 10 raised to the dth power.
	FRACTw. Format (p. 142)	Converts numeric values to fractions.

Category	Language Elements	Description
	HEXw. Format (p. 143)	Converts real binary (floating-point) values to hexadecimal representation.
	IEEEw.d Format (p. 148)	Generates an IEEE floating-point value by multiplying a number by 10 raised to the d th power.
	NEGPARENw.d Format (p. 162)	Writes negative numeric values in parentheses.
	OCTALw. Format (p. 165)	Converts numeric values to octal representation.
	PERCENTw.d Format (p. 166)	Writes numeric values as percentages.
	PERCENTNw.d Format (p. 168)	Produces percentages, using a minus sign for negative values.
	ROMANw. Format (p. 171)	Writes numeric values as roman numerals.
	SIZEKw.d Format (p. 172)	Writes a numeric value in the form nK for kilobytes.
	VAXRBw.d Format (p. 178)	Writes real binary (floating-point) data in VMS format.
	w.d Format (p. 179)	Writes standard numeric data one digit per byte.
	YENw.d Format (p. 186)	Writes numeric values with yen signs, commas, and decimal points.
	Zw.d Format (p. 204)	Writes standard numeric data with leading 0s.

Dictionary

$BASE64Xw. Format

Converts character data into ASCII text by using Base 64 encoding.

Categories: Character
CAS

Alignment: Left

Syntax

$BASE64X*w*.

Arguments

w

specifies the width of the output field. You can use the following formula to determine the width:`format-width=(variable-length+2)/3*4`

When the variable-length+2 is divided by 3, the results are truncated to an integer and multiplied by 4. For example, if a variable length is 28, as for the input value `www.mydomain.com/myhiddenURL` in the examples below, the width calculation is (28+2)/3*4=40. If the format width is too small, the value is not converted. No message is written to the SAS log.

Default 1

Range 1–32767

Note The default width of 1 is insufficient to produce a formatted value. Do not specify $BASE64X without a width.

Details

Base 64 is an industry encoding method whose encoded characters are determined by using a positional scheme that uses only ASCII characters. Several Base 64 encoding schemes have been defined by the industry for specific uses, such as email or content masking. SAS maps positions 0–61 to the characters A–Z, a–z, and 0–9. Position 62 maps to the character +, and position 63 maps to the character /.

Here are some uses of Base 64 encoding:

- embed binary data in an XML file
- encode passwords
- encode URLs

The '=' character in the encoded results indicates that the results have been padded with zero bits. In order for the encoded characters to be decoded, the '=' must be included in the value to be decoded.

Example

The following examples illustrate the BASE64X*w.* format:

Statements	Results
`select put ('FCA01A7993BC', $base64x64.);`	`RkNBMDFBNzk5M0JD`

Statements	Results
`select put ('MyPassword', $base64x64.);`	`TXlQYXNzd29yZA==`
`select put ('www.mydomain.com/myhiddenURL', $base64x64.);`	`d3d3Lm15ZG9tYWluLmNvbi9teWhpZGRlblVSTA==`

$BINARYw. Format

Converts character data to binary representation.

Categories: Character
CAS

Alignment: Left

Syntax

$BINARY*w.*

Arguments

w
specifies the width of the output field.

Default The default width is calculated based on the length of the variable to be printed.

Range 1–32767

Comparisons

The $BINARY*w*. format converts character values to binary representation. The BINARY*w*. format converts numeric values to binary representation.

Example

The example table uses the input value AB to illustrate the effect of the $BINARY*w.* format.

Statements	Results	
`select put('AB', $binary16.);`	ASCII	EBCDIC
	`0100000101000010`	`1100000111000010`

See Also

Formats:

- "BINARYw. Format" on page 110
- "$HEXw. Format" on page 98

$CHARw. Format

Writes standard character data.

Categories: Character
CAS

Alignment: Left

Alias: $*w*. and $F*w*.

Syntax

$CHAR*w.*

Arguments

w

specifies the width of the output field.

Default 8 if the length of variable is undefined; otherwise, the length of the variable.

Range 1–32767

Comparisons

The $CHAR*w*., $*w*., and $F*w*. formats are identical. They do not trim leading blanks.

Example

The example table uses the input value XYZ to illustrate the effect of the $CHAR*w.* format.

Statements	Results
`select put ('XYZ',$char1.);`	`X`
`select put ('XYZ',$char2.);`	`XY`
`select put ('XYZ',$char.);`	`XYZ`

See Also

Formats:

- "$w. Format" on page 105

$HEXw. Format

Converts character data to hexadecimal representation.

Categories: Character
CAS

Alignment: Left

Syntax

$HEX*w.*

Arguments

w

specifies the width of the output field.

Default The default width is calculated based on the length of the variable to be printed.

Range 1–32767

Tips To ensure that SAS writes the full hexadecimal equivalent of your data, make *w* twice the length of the variable or field that you want to represent.

If *w* is greater than twice the length of the variable that you want to represent, $HEX*w*. pads it with blanks.

Details

The $HEX*w*. format converts each character into two hexadecimal characters. Each blank counts as one character, including trailing blanks.

Comparisons

The HEX*w*. format converts real binary numbers to their hexadecimal equivalent.

Example

The following example illustrates the $HEX*w*. format:

Statements	Results
`select put ('AB',$hex5.);`	`4142`

See Also

Formats:

- "$BINARYw. Format" on page 96
- "HEXw. Format" on page 143

$OCTALw. Format

Converts character data to octal representation.

Categories: Character
CAS

Alignment: Left

Syntax

$OCTAL*w.*

Arguments

w

specifies the width of the output field.

Default The default width is calculated based on the length of the variable to be printed.

Range 1–32767

Tip Because each character value generates three octal characters, increase the value of *w* by three times the length of the character value.

Comparisons

The $OCTAL*w*. format converts character values to the octal representation of their character codes. The OCTAL*w*. format converts numeric values to octal representation.

Example

The following examples illustrate the $OCTAL*w.* format:

Statements	Results
`select put ('art', $octal15.);`	`141162164`
`select put ('rice', $octal15.);`	`162151143145`
`select put ('bank', $octal15.);`	`142141156153`

See Also

Formats:

- “OCTALw. Format” on page 165

$QUOTEw. Format

Writes data values that are enclosed in single quotation marks.

Categories: Character
CAS

Alignment: Left

Syntax

$QUOTE*w.*

Arguments

w
specifies the width of the output field.

Default 2 if the length of the variable is undefined; otherwise, the length of the variable + 2.

Range 2–32767

Tip Make *w* wide enough to include the left and right quotation marks.

Details

When you use the $QUOTE*w*. format, all literals must be in single quotation marks.

The following list describes the output that SAS produces when you use the $QUOTEw. format.

- When your data value is not enclosed in quotation marks, SAS encloses the output in double quotation marks.
- When your data value is not enclosed in quotation marks, but the value contains a single quotation mark, SAS takes the following actions:
  - encloses the data value in double quotation marks
  - does not change the single quotation mark.
- When your data value begins and ends with single quotation marks, and the value contains double quotation marks, SAS takes the following actions:
  - encloses the data value in double quotation marks
  - duplicates the double quotation marks that are found in the data value
  - does not change the single quotation marks.

- When your data value begins and ends with single quotation marks, and the value contains two single contiguous quotation marks, SAS takes the following actions:
  - encloses the value in double quotation marks
  - does not change the single quotation marks.
- When your data value begins and ends with single quotation marks, and contains both double quotation marks and single, contiguous quotation marks, SAS takes the following actions:
  - encloses the value in double quotation marks
  - duplicates the double quotation marks that are found in the data value
  - does not change the single quotation marks.
- When the length of the target field is not large enough to contain the string and its quotation marks, SAS returns all blanks.

Note: An embedded quotation mark must be enclosed within additional quotation marks. If you specify a value of A"B, then $QUOTE5. truncates the B in the value and writes "A"""".

Example

The following examples illustrate the $QUOTE*w.* format:

Statements	Results
`select put('SAS',$quote.);`	`"SAS"`
`select put('SAS''s',$quote.);`	`"SAS's"`
`select put('ad''verb',$quote16.);`	`"ad'verb"`
`select put('"ad"'''"verb"',$quote20.);`	`"""ad""'""verb"""`

$REVERJw. Format

Writes character data in reverse order and preserves blanks.

Categories: Character
CAS

Alignment: Right

Syntax

$REVERJ*w*.

Arguments

w

specifies the width of the output field.

Default 1 if *w* is not specified

Range 1–32767

Comparisons

The $REVERJ*w*. format is similar to the $REVERS*w*. format except that $REVERS*w*. left aligns the result by trimming all leading blanks.

Example

The following examples illustrate the $REVERJ*w*. format:

Statements	Results
`select put('ABCD###',$reverj7.);`	`###DCBA`
`select put('###ABCD',$reverj7.);`	`DCBA###`

See Also

Formats:

- "$REVERSw. Format" on page 103

$REVERSw. Format

Writes character data in reverse order and left aligns.

Categories: Character
CAS

Alignment: Left

Syntax

$REVERS*w*.

Arguments

w

specifies the width of the output field.

Default 1 if *w* is not specified

Range 1–32767

Comparisons

The $REVERS*w*. format is similar to the $REVERJ*w*. format except that $REVERJ*w*. does not left align the result.

Example

The following examples illustrate the $REVERS*w.* format:

Statements	Results
`select put('ABCD###',$revers7.);`	`###DCBA`
`select put('###ABCD',$revers7.);`	`DCBA###`

See Also

Formats:

- "$REVERJw. Format" on page 102

$UPCASEw. Format

Converts character data to uppercase.

Categories: Character
CAS

Alignment: Left

Syntax

$UPCASE*w.*

Arguments

w

specifies the width of the output field.

Default 8 if the length of the variable is undefined; otherwise, the length of the variable.

Range 1–32767

Details

Special characters, such as hyphens and other symbols, are not altered.

Example

The following example illustrates the $UPCASE*w.* format:

Statements	Results
`select put('coxe-ryan',$upcase9.);`	`COXE-RYAN`

$w. Format

Writes standard character data.

Categories: Character
CAS

Alignment: Left

Alias: $F*w*.

Syntax

$*w.*

Arguments

w

specifies the width of the output field.

Default 1 if the length of the identifier is undefined; otherwise, the length of the identifier

Range 1–32767

Comparisons

The $*w*. format and the $CHAR*w*. format are identical, and they do not trim leading blanks.

Example

The following examples illustrate the $*w*. format:

Statements[1]	Results[1]
`select put('#Cary',$5.);`	`#Cary`
`select put('#Cary',$f5.);`	`#Cary`
`select put('Carolina',$5.);`	`Carol`

1 The character # represents a blank space.

See Also

Formats:

- "$CHARw. Format" on page 97
- "w.d Format" on page 179

BESTw. Format

SAS chooses the best notation.

Categories: Numeric
CAS

Alignment: Right

Syntax

BEST*w.*

Arguments

w

specifies the width of the output field.

Default 12

Range 1–32

Tip If you print numbers between 0 and .01 exclusively, use a field width of at least 7 to avoid excessive rounding. If you print numbers between 0 and –.01 exclusively, use a field width of at least 8.

Details

The BEST*w*. format is the default format for writing numeric values. When there is no format specification, SAS chooses the format that provides the most information about the value according to the available field width. BEST*w*. rounds the value, and if SAS can display at least one significant digit in the decimal portion, within the width specified, BEST*w*. produces the result in decimal. Otherwise, it produces the result in scientific notation. SAS always stores the complete value regardless of the format that you use to represent it.

Comparisons

- The BEST*w.* format writes as many significant digits as possible in the output field, but if the numbers vary in magnitude, the decimal points do not line up. Integers are printed without a decimal.
- The D*w.p* format writes numbers with the desired precision and more alignment than the BEST*w.* format.

Example

The example table uses the value 1257000 to illustrate the effect of the BEST*w.* format.

Statements	Results
`select put(1257000,best6.);`	`1.26E6`
`select put(1257000,best3.);`	`1E6`

See Also

Formats:

- "BESTDw.p Format" on page 108
- "Dw.p Format" on page 114
- "w.d Format" on page 179

BESTDw.p Format

Prints numeric values, lining up decimal places for values of similar magnitude, and prints integers without decimals.

Categories: Numeric
CAS

Note: In SAS 9.4, the output of the BESTDw. format is right-aligned; in CAS, the output is left-aligned.

Syntax

BESTD*w*.[*p*]

Arguments

w

specifies the width of the output field.

Default 12

Range 1–32

p

specifies the precision.

Default 3

Range 0 to *w*–1

Requirement must be less than *w*

Tip If *p* is omitted or is specified as 0, then *p* is set to 3.

Details

The BESTD*w.p* format writes numbers so that the decimal point aligns in groups of values with similar magnitude. Integers are printed without a decimal point. Larger values of *p* print the data values with more precision and potentially more shifts in the decimal point alignment. Smaller values of *p* print the data values with less precision and a greater chance of decimal point alignment.

The format chooses the number of decimal places to print for ranges of values, even when the underlying values can be represented with fewer decimal places.

Comparisons

- The BEST*w*. format writes as many significant digits as possible in the output field, but if the numbers vary in magnitude, the decimal points do not line up. Integers are printed without a decimal.
- The D*w.p* format writes numbers with the desired precision and more alignment than the BEST*w* format.
- The BESTD*w.p* format is a combination of the BEST*w*. format and the D*w.p* format in that it formats all numeric data, and it does a better job of aligning decimals than the BEST*w*. format.
- The *w.d* format aligns decimal points, if possible, but it does not necessarily show the same precision for all numbers.

Example

The following examples illustrate the BESTD*w.p* format:

Statements	Results
`select put (12345, bestd14.);`	`12345`
`select put (123.45, bestd14.);`	`123.4500000`
`select put (1.2345, bestd14.);`	`1.2345000`
`select put (.12345, bestd14.);`	`0.1234500`
`select put (1.23456789, bestd14.);`	`1.23456789`

See Also

Formats:

- "BESTw. Format" on page 106
- "Dw.p Format" on page 114
- "w.d Format" on page 179

BINARYw. Format

Converts numeric values to binary representation.

Categories: Numeric
CAS

Alignment: Left

Syntax

BINARY*w.*

Arguments

w

specifies the width of the output field.

Default 8

Range 1–64

Comparisons

BINARY*w*. converts numeric values to binary representation. The $BINARY*w*. format converts character values to binary representation.

Example

The following examples illustrate the BINARY*w.* format:

Statements	Results
`select put (123.45, binary8.);`	`01111011`
`select put (123, binary8.);`	`01111011`
`select put (-123, binary8.);`	`10000101`

See Also

Formats:

- "$BINARYw. Format" on page 96
- "HEXw. Format" on page 143

COMMAw.d Format

Writes numeric values with a comma that separates every three digits and a period that separates the decimal fraction.

Categories:	Numeric CAS
Alignment:	Right

Syntax

COMMA*w*.[*d*]

Arguments

w

specifies the width of the output field.

Default 6

Range 1–32

Tip Make *w* wide enough to write the numeric values, the commas, and the optional decimal point.

d

specifies the number of digits to the right of the decimal point in the numeric value.

Range 0–31

Requirement must be less than *w*

Comparisons

- The COMMA*w.d* format is similar to the COMMAX*w.d* format, but the COMMAX*w.d* format reverses the roles of the decimal point and the comma. This convention is common in European countries.
- The COMMA*w.d* format is similar to the DOLLAR*w.d* format except that the COMMA*w.d* format does not print a leading dollar sign.

Example

The following examples illustrate the COMMA*w.d* format:

Statements	Results
`select put (23451.23,comma10.2);`	`23,451.23`
`select put (123451.234,comma10.2);`	`123,451.23`

See Also

Formats:

- "COMMAXw.d Format" on page 112
- "DOLLARw.d Format" on page 126

COMMAXw.d Format

Writes numeric values with a period that separates every three digits and a comma that separates the decimal fraction.

Categories: Numeric
CAS

Alignment: Right

Syntax

COMMAX[*w*].*d*

Arguments

w

specifies the width of the output field.

Default 6

Range 1–32

Tip Make *w* wide enough to write the numeric values, the commas, and the optional decimal point.

d

specifies the number of digits to the right of the decimal point in the numeric value.

Range 0–31

Requirement must be less than *w*

Comparisons

The COMMA*w.d* format is similar to the COMMAX*w.d* format, but the COMMAX*w.d* format reverses the roles of the decimal point and the comma. This convention is common in European countries.

Example

The following examples illustrate the COMMAX*w.d* format:

Statements	Results
`select put (23451.23,commax10.2);`	`23.451,23`
`select put (123451.234,commax10.2);`	`123.451,23`

See Also

Formats:

- "COMMAw.d Format" on page 111
- "DOLLARXw.d Format" on page 127

Dw.p Format

Prints variables, possibly with a great range of values, lining up decimal places for values of similar magnitude.

Categories: Numeric
CAS

Alignment: Right

Syntax

D[*w*].[*p*]

Arguments

w
specifies the width of the output field.

Default 12

Range 1–32

p
specifies the significant digits.

Default 3

Range 0–16

Requirement must be less than *w*

Details

The D*w.p* format writes numbers so that the decimal point aligns in groups of values with similar magnitude. Larger values of *p* print the data values with more precision and potentially more shifts in the decimal point alignment. Smaller values of p print the data values with less precision and a greater chance of decimal point alignment.

Comparisons

- The BEST*w*. format writes as many significant digits as possible in the output field, but if the numbers vary in magnitude, the decimal points do not line up.

- D*w.p* writes numbers with the desired precision and more alignment than BEST*w*.
- The BESTD*w.p* format is a combination of the BEST*w*. format and the D*w.p* format in that it formats all numeric data, and it does a better job of aligning decimals than the BEST*w*. format.
- The *w.d* format aligns decimal points, if possible, but it does not necessarily show the same precision for all numbers.

Example

The following examples illustrate the D*w.p* format:

Statements	Results
`select put (12345,d10.4);`	`12345.0`
`select put (1234.5,d10.4);`	`1234.5`
`select put (123.45,d10.4);`	`123.45000`
`select put (12.345,d10.4);`	`12.34500`
`select put (1.2345,d10.4);`	`1.23450`
`select put (.12345,d10.4);`	`0.12345`

See Also

Formats:

- “BESTw. Format” on page 106
- “BESTDw.p Format” on page 108
- “w.d Format” on page 179

DATEw. Format

Writes SAS date values in the form *ddmmmyy*, *ddmmmyyyy*, or *dd-mmm-yyyy*.

Categories: Date and Time
CAS

Alignment: Right

Syntax

DATE*w.*

Arguments

w
specifies the width of the output field.

Default 7

Range 5–9

Tip Use a width of 9 to print a 4-digit year.

Details

The DATE*w*. format writes SAS date values in the form *ddmmmyy*, *ddmmmyyyy*, or *dd-mmm-yyyy* where

dd
is an integer that represents the day of the month.

mmm
is the first three letters of the month name.

yy or *yyyy*
is a two-digit or four-digit integer that represents the year.

Note: SAS calculates date values based on the number of days since January 1, 1960.

Example

The example table uses the input value of 19431, which is the SAS date value that corresponds to March 14, 2013.

Statements	Results
`select put (19431,date5.);`	14MAR
`select put (19431,date6.);`	14MAR
`select put (19431,date7.);`	14MAR13
`select put (19431,date8.);`	14MAR13

Statements	Results
`select put (19431,date9.);`	`14MAR2013`

See Also

Formats:

- “DATETIMEw.d Format” on page 119
- “DAYw. Format” on page 121
- “DDMMYYw. Format” on page 122
- “DTDATEw. Format” on page 130
- “JULIANw. Format” on page 149
- “MMDDYYw. Format” on page 151
- “MONTHw. Format” on page 160
- “NENGOw. Format” on page 164
- “WEEKDATEw. Format” on page 181
- “YEARw. Format” on page 185

Functions:

- “CURRENT_DATE Function” on page 406
- “DATEPART Function” on page 418
- “DAY Function” on page 420
- “MAKEDATE Function” on page 633

DATEAMPMw.d Format

Writes SAS datetime values in the form *ddmmmyy:hh:mm:ss.ss* with AM or PM.

Categories: Date and Time
CAS

Alignment: Right

Syntax

DATEAMPM*w*.[*d*]

Arguments

w

specifies the width of the output field.

Default 19

Range 7–40

Tip SAS requires a minimum *w* value of 13 to write AM or PM. For widths between 10 and 12, SAS writes a 24-hour clock time.

d

specifies the number of digits to the right of the decimal point in the seconds value.

Range 0–39

Requirement must be less than *w*

Note If *w* – *d*< 17, SAS truncates the decimal values.

Details

The DATEAMPM*w.d* format writes SAS datetime values in the form *ddmmmyy:hh:mm:ss.ss*, where

dd
is an integer that represents the day of the month.

mmm
is the first three letters of the month name.

yy
is a two-digit integer that represents the year.

hh
is an integer that represents the hour.

mm
is an integer that represents the minutes.

ss.ss
is the number of seconds to two decimal places.

Comparisons

The DATETIME*w.d* format is similar to the DATEAMPM*w.d* format except that DATETIME*w.d* does not print AM or PM at the end of the time.

Example

The example table uses the input value of 1679344494, which is the SAS datetime value that corresponds to 08:34:54 PM on March 19, 2013.

Statements	Results
`select put (1679344494,dateampm.);`	`19MAR13:08:34:54 PM`
`select put (1679344494,dateampm7.);`	`19MAR13`
`select put (1679344494,dateampm10.);`	`19MAR13:20`
`select put (1679344494,dateampm13.);`	`19MAR13:08 PM`
`select put (1679344494,dateampm22.2.);`	`19MAR13:08:34:54.00 PM`

See Also

Formats:

- “DATETIMEw.d Format” on page 119
- “TIMEAMPMw.d Format” on page 175

DATETIMEw.d Format

Writes SAS datetime values in the form *ddmmmyy:hh:mm:ss.ss*.

Categories: Date and Time
CAS

Alignment: Right

Syntax

DATETIME*w*.[*d*]

Arguments

w

specifies the width of the output field.

Default 16

Range 7–40

Tips SAS requires a minimum *w* value of 16 to write a SAS datetime value with the date, hour, and seconds. Add an additional two places to *w* and a value to *d* to return values with optional decimal fractions of seconds.

If *w* – *d*< 17, SAS truncates the decimal values.

d
specifies the number of digits to the right of the decimal point in the seconds value.

Range 0–39

Requirement must be less than *w*

Details

The DATETIME*w.d* format writes SAS datetime values in the form *ddmmmyy:hh:mm:ss.ss*, where

dd
is an integer that represents the day of the month.

mmm
is the first three letters of the month name.

yy
is a two-digit integer that represents the year.

hh
is an integer that represents the hour in 24-hour clock time.

mm
is an integer that represents the minutes.

ss.ss
is the number of seconds to two decimal places.

Comparisons

The DATEAMPM*w.d* format is similar to the DATETIME*w.d* format except that DATEAMPM*w.d* prints AM or PM at the end of the time.

Example

The example table uses the input value of 1699674559, which is the SAS datetime value that corresponds to 3:49:19 AM on November 10, 2013.

Statements	Results
`select put (1699674559,datetime.);`	`10NOV13:03:49:19`
`select put (1699674559,datetime7.);`	`10NOV13`
`select put (1699674559,datetime12.);`	`10NOV13:03`
`select put (1699674559,datetime18.);`	`10NOV13:03:49:19`
`select put (1699674559,datetime18.1);`	`10NOV13:03:49:19.0`
`select put (1510285759,datetime19.);`	`10NOV2013:03:49:19`
`select put (1699674559,datetime20.1);`	`10NOV2013:03:49:19.0`
`select put (1699674559,datetime21.2);`	`10NOV2013:03:49:19.00`

See Also

Formats:

- “DATEAMPMw.d Format” on page 117
- “DDMMYYxw. Format” on page 124
- “MMDDYYw. Format” on page 151

DAYw. Format

Writes SAS date values as the day of the month.

Categories: Date and Time
CAS

Alignment: Right

Syntax

DAY*w.*

Arguments

w
specifies the width of the output field.

Default 2

Range 2–32

Example

The example table uses the input value of 19523, which is the SAS date value that corresponds to June 14, 2013.

Statements	Results
`select put(19523,day2.);`	`14`

See Also

Formats:

- "DATEw. Format" on page 115

Functions:

- "DAY Function" on page 420

DDMMYYw. Format

Writes SAS date values in the form *ddmm*[*yy*]*yy* or *dd*/*mm*/[*yy*]*yy*, where a forward slash is the separator and the year appears as either 2 or 4 digits.

Categories: Date and Time
CAS

Alignment: Right

Syntax

DDMMYY*w.*

Arguments

w
specifies the width of the output field.

Default 8

Range 2–10

Interaction When *w* has a value of from 2 to 5, the date appears with as much of the day and the month as possible. When *w* is 7, the date appears as a two-digit year without slashes.

Details

The DDMMYY*w*. format writes SAS date values in the form *ddmm*[*yy*]*yy* or *dd*/*mm*/[*yy*]*yy*, where

dd
is an integer that represents the day of the month.

/
is the separator.

mm
is an integer that represents the month.

[*yy*]*yy*
is a two-digit or four-digit integer that represents the year.

Example

The following examples use the input value of 19704, which is the SAS date value that corresponds to December 12, 2013.

Statements	Results
`select put(19704,ddmmyy.);`	12/12/13
`select put(19704,ddmmyy5.);`	12/12
`select put(19704,ddmmyy6.);`	121213
`select put(19704,ddmmyy7.);`	121213
`select put(19704,ddmmyy8.);`	12/12/13
`select put(19704,ddmmyy10.);`	12/12/2013

See Also

Formats:

- "DATEw. Format" on page 115
- "DDMMYYxw. Format" on page 124
- "MMDDYYw. Format" on page 151
- "YYMMDDw. Format" on page 191

Functions:

- "DAY Function" on page 420
- "MONTH Function" on page 655
- "YEAR Function" on page 864

DDMMYYxw. Format

Writes SAS date values in the form *ddmm*[*yy*]*yy* or *dd*X*mm*X[*yy*]*yy*, where X represents a specified separator and the year appears as either 2 or 4 digits.

Categories: Date and Time
CAS

Alignment: Right

Syntax

DDMMYY*xw.*

Arguments

x

identifies a separator or specifies that no separator appear between the day, the month, and the year. Valid values for *x* are any of the following:

B
separates with a blank

C
separates with a colon

D
separates with a hyphen

N
indicates no separator

P
separates with a period

S
separates with a slash.

w

specifies the width of the output field.

Default 8

Range 2–10

Interactions When *w* has a value of from 2 to 5, the date appears with as much of the day and the month as possible. When *w* is 7, the date appears as a two-digit year without separators.

When *x* has a value of N, the width range changes to 2–8.

Details

The DDMMYY*xw*. format writes SAS date values in the form *ddmm*[*yy*]*yy* or *dd*X*mm*X[*yy*]*yy*, where

dd
is an integer that represents the day of the month.

X
is a specified separator.

mm
is an integer that represents the month.

[*yy*]*yy*
is a two-digit or four-digit integer that represents the year.

Example

The following examples use the input value of 19431, which is the SAS date value that corresponds to March 14, 2013.

Statements	Results
`select put(19431,ddmmyyc5.);`	`14:03`
`select put(19431,ddmmyyd8.);`	`14-03-13`
`select put(19431,ddmmyyn8.);`	`14032013`
`select put(19431,ddmmyyp10.);`	`14.03.2013`

See Also

Formats:

- “DATEw. Format” on page 115

- “DDMMYYw. Format” on page 122
- “MMDDYYxw. Format” on page 152
- “YYMMDDxw. Format” on page 193

Functions:

- “DAY Function” on page 420
- “MONTH Function” on page 655
- “YEAR Function” on page 864

DOLLARw.d Format

Writes numeric values with a leading dollar sign, a comma that separates every three digits, and a period that separates the decimal fraction.

Categories: Numeric
CAS

Alignment: Right

Syntax

DOLLAR*w*.[*d*]

Arguments

w

specifies the width of the output field.

Default 6

Range 2–32

d

specifies the number of digits to the right of the decimal point in the numeric value.

Range 0–31

Requirement must be less than *w*

Details

The DOLLAR*w.d* format writes numeric values with a leading dollar sign, a comma that separates every three digits, and a period that separates the decimal fraction.

The hexadecimal representation of the code for the dollar sign character ($) is 5B on EBCDIC systems and 24 on ASCII systems. The monetary character that these codes represent might be different in other countries, but DOLLAR*w.d* always produces one of these codes.

Comparisons

- The DOLLAR*w.d* format is similar to the DOLLARX*w.d* format, but the DOLLARX*w.d* format reverses the roles of the decimal point and the comma. This convention is common in European countries.
- The DOLLAR*w.d* format is the same as the COMMA*w.d* format except that the COMMA*w.d* format does not write a leading dollar sign.

Example

The following example illustrates the DOLLAR*w.d* format:

Statements	Results
`select put(1254.71,dollar10.2);`	`$1,254.71`

See Also

Formats:

- “COMMAw.d Format” on page 111
- “DOLLARXw.d Format” on page 127
- “EUROw.d Format” on page 138

DOLLARXw.d Format

Writes numeric values with a leading dollar sign, a period that separates every three digits, and a comma that separates the decimal fraction.

Categories: Numeric
CAS

Alignment: Right

Syntax

DOLLARX*w*.[*d*]

Arguments

w

specifies the width of the output field.

Default 6

Range 2–32

d

specifies the number of digits to the right of the decimal point in the numeric value.

Default 0

Range 2–31

Requirement must be less than *w*

Details

The DOLLARX*w.d* format writes the numeric values with a leading dollar sign, a comma that separates every three digits, and a period that separates the decimal fraction.

The hexadecimal representation of the code for the dollar sign character ($) is 5B on EBCDIC systems and 24 on ASCII systems. The monetary character that these codes represent might be different in other countries, but DOLLARX*w.d* always produces one of these codes.

Comparisons

- The DOLLARX*w.d* format is similar to the DOLLAR*w.d* format, but the DOLLARX*w.d* format reverses the roles of the decimal point and the comma. This convention is common in European countries.
- The DOLLARX*w.d* format is the same as the COMMAX*w.d* format except that the COMMA*w.d* format does not write a leading dollar sign.

Example

The following example illustrates the DOLLARX*w.d* format:

Statements	Results
`select put(1254.71,dollarx10.2);`	`$1.254,71`

See Also

Formats:

- “COMMAXw.d Format” on page 112
- “DOLLARw.d Format” on page 126

DOWNAMEw. Format

Writes SAS date values as the name of the day of the week.

Categories: Date and Time
CAS

Alignment: Right

Syntax

DOWNAME*w.*

Arguments

w

specifies the width of the output field.

Default 9

Range 1–32

Tip If you omit *w*, SAS prints the entire name of the day.

Example

The example table uses the input value of 19431, which is the SAS date value that corresponds to March 14, 2013.

Statements	Results
`select put(19431,downame.);`	`Thursday`

See Also

Formats:

- “DATEw. Format” on page 115
- “DTWKDATXw. Format” on page 133
- “WEEKDATEw. Format” on page 181
- “WEEKDATXw. Format” on page 182
- “WEEKDAYw. Format” on page 184

DTDATEw. Format

Expects a SAS datetime value as input and writes the SAS date values in the form *ddmmmyy* or *ddmmmyyyy*.

Categories: Date and Time
CAS

Alignment: Right

Syntax

DTDATE*w.*

Arguments

w

specifies the width of the output field.

Default 7

Range 5–9

Tip Use a width of 9 to print a 4-digit year.

Details

The DTDATE*w*. format writes SAS date values in the form *ddmmmyy* or *ddmmmyyyy*, where

dd
is an integer that represents the day of the month.

mmm
are the first three letters of the month name.

yy or *yyyy*
is a two-digit or four-digit integer that represents the year.

Comparisons

The DTDATE*w*. format produces the same type of output that the DATE*w*. format produces. The difference is that the DTDATE*w*. format requires a SAS datetime value.

Example

The example table uses the input value of 1823167525, which is the SAS datetime value that corresponds to 11:25:25 AM on October 9, 2017, and prints both a two-digit and a four-digit year for the DTDATE*w*. format.

Statements	Results
`select put(1823167525,dtdate.);`	`09OCT17`
`select put(1823167525,dtdate9.);`	`09OCT2017`

See Also

Formats:

- “DATEw. Format” on page 115

DTMONYYw. Format

Writes the date part of a SAS datetime value as the month and year in the form *mmmyy* or *mmmyyyy*.

Categories: Date and Time
CAS

Alignment: Right

Syntax

DTMONYY*w.*

Arguments

w
specifies the width of the output field.

Default 5

Range 5–7

Details

The DTMONYY*w*. format writes SAS datetime values in the form *mmmyy* or *mmmyyyy*, where

mmm
is the first three letters of the month name.

yy or *yyyy*
is a two-digit or four-digit integer that represents the year.

Comparisons

The DTMONYY*w*. format and the MONYY*w*. format are similar in that they both write date values. The difference is that DTMONYY*w*. expects a SAS datetime value as input, and MONYY*w*. expects a SAS date value.

Example

The example table uses as input the value 1823167525, which is the SAS datetime value that corresponds to October 9, 2017, at 11:25:25 a.m.

Statements	Results
`select put(1823167525,dtmonyy.);`	`2017:4`
`select put(1823167525,dtmonyy5.);`	`OCT17`
`select put(1823167525,dtmonyy6.);`	`OCT17`

Statements	Results
`select put(1823167525,dtmonyy7.);`	`OCT2017`

See Also

Formats:

- “DATETIMEw.d Format” on page 119
- “MONYYw. Format” on page 161

DTWKDATXw. Format

Writes the date part of a SAS datetime value as the day of the week and the date in the form *day-of-week*, *dd month-name yy* (or *yyyy*).

Categories: Date and Time
CAS

Alignment: Right

Syntax

DTWKDATX*w*.

Arguments

w
specifies the width of the output field.

Default 29

Range 3–37

Details

The DTWKDATX*w*. format writes SAS date values in the form *day-of-week*, *dd month-name*, *yy* or *yyyy*, where

day-of-week
is either the first three letters of the day name or the entire day name.

dd
is an integer that represents the day of the month.

month-name
is either the first three letters of the month name or the entire month name.

yy or *yyyy*
is a two-digit or four-digit integer that represents the year.

Comparisons

The DTWKDATX*w*. format is similar to the WEEKDATX*w*. format in that they both write date values. The difference is that DTWKDATX*w*. expects a SAS datetime value as input, and WEEKDATX*w*. expects a SAS date value.

Example

The example table uses as input the value 1823167525, which is the SAS datetime value that corresponds to October 9, 2017, at 11:25:25 a.m.

Statements	Results
`select put(1823167525,dtwkdatx.);`	`Monday, 9 October 2017`
`select put(1823167525,dtwkdatx3.);`	`Mon`
`select put(1823167525,dtwkdatx8.);`	`Mon`
`select put(1823167525,dtwkdatx25.);`	`Monday, 9 Oct 2017`

See Also

Formats:

- “DATEw. Format” on page 115
- “WEEKDATEw. Format” on page 181
- “WEEKDATXw. Format” on page 182

DTYEARw. Format

Writes the date part of a SAS datetime value as the year in the form *yy* or *yyyy*.

Categories: Date and Time

CAS

Alignment: Right

Syntax

DTYEAR*w*.

Arguments

w

specifies the width of the output field.

Default 4

Range 2–4

Details

remove after conversion

Comparisons

The DTYEAR*w*. format is similar to the YEAR*w*. format in that they both write date values. The difference is that DTYEAR*w*. expects a SAS datetime value as input, and YEAR*w*. expects a SAS date value.

Example

The example table uses as input the value 1823167525, which is the SAS datetime value that corresponds to October 9, 2017, at 11:25:25 a.m.

Statements	Results
`select put(1823167525,dtyear.);`	2017
`select put(1823167525,dtyear2.);`	17
`select put(1823167525,dtyear3.);`	17
`select put(1823167525,dtyear4.);`	2017

See Also

Formats:

- "DATEw. Format" on page 115
- "DATETIMEw.d Format" on page 119
- "YEARw. Format" on page 185

DTYYQCw. Format

Writes the date part of a SAS datetime value as the year and the quarter, and separates them with a colon (:).

Categories:	Date and Time CAS
Alignment:	Right

Syntax

DTYYQC*w.*

Arguments

w

specifies the width of the output field.

Default 4

Range 4–6

Details

The DTYYQC*w*. format writes SAS datetime values in the form *yy* or *yyyy*, followed by a colon (:) and the numeric value for the quarter of the year.

Example

The example table uses as input the value 1823167525, which is the SAS datetime value that corresponds to October 9, 2017, at 11:25:25 a.m.

Statements	Results
	`----+----1`
`select put(1823167525,dtyyqc.);`	`17:4`
`select put(1823167525,dtyyqc4.);`	`17:4`
`select put(1823167525,dtyyqc5.);`	`17:4`
`select put(1823167525,dtyyqc6.);`	`2017:4`

See Also

Formats:

- “DATEw. Format” on page 115
- “DATETIMEw.d Format” on page 119

Ew. Format

Writes numeric values in scientific notation.

Categories: Numeric
CAS

Alignment: Right

Syntax

E*w.*

Arguments

w
specifies the width of the output field.

Default 12

Range 7–32

Details

SAS reserves the first column of the result for a minus sign.

Example

The following examples illustrate the e*w.* format:

Statements	Results
`select put(1257,e10.);`	`1.257E+03`
`select put(-1257,e10.);`	`-1.257E+03`

EUROw.d Format

Writes numeric values with a leading euro symbol (€), a comma that separates every three digits, and a period that separates the decimal fraction.

Categories: Numeric
CAS

Alignment: Right

Syntax

EURO*w.d*

Arguments

w

specifies the width of the output field.

Default 6

Range 1–32

Tip If you want the euro symbol to be part of the output, be sure to choose an adequate width.

d

specifies the number of digits to the right of the decimal point in the numeric value.

Default 0

Range 0–31

Requirement must be less than *w*

Comparisons

- The EURO*w.d* format is similar to the EUROX*w.d* format, but EUROX*w.d* format reverses the roles of the decimal point and the comma. This convention is common in European countries.
- The EURO*w.d* format is similar to the DOLLAR*w.d* format, except that DOLLAR*w.d* format writes a leading dollar sign instead of the euro symbol.

Note: The EURO*w.d* format uses the euro character (U+20AC). If you use the DBCS version of SAS and an encoding that does not support the euro character, an error will occur. To prevent this error, change your session encoding to an encoding that supports the euro character.

Example

These examples use 1254.71 as the value of amount.

Statements	Results
`select put(1254.71,euro10.2);`	`E1,254.71`
`select put(1254.71,euro5.);`	`1,255`
`select put(1254.71,euro9.2);`	`E1,254.71`
`select put(1254.71,euro15.3);`	`E1,254.710`

See Also

Formats:

- “DOLLARw.d Format” on page 126
- “EUROXw.d Format” on page 140
- “YENw.d Format” on page 186

EUROXw.d Format

Writes numeric values with a leading euro symbol (€), a period that separates every three digits, and a comma that separates the decimal fraction.

Categories: Numeric
CAS

Alignment: Right

Syntax

EUROX*w.d*

Arguments

w

specifies the width of the output field.

Default 6

Range 1–32

Tip If you want the euro symbol to be part of the output, be sure to choose an adequate width.

d

specifies the number of digits to the right of the decimal point in the numeric value.

Default 0

Range 0–31

Requirement must be less than *w*

Comparisons

- The EUROX*w.d* format is similar to the EURO*w.d* format, but EURO*w.d* format reverses the roles of the comma and the decimal point. This convention is common in English-speaking countries.
- The EUROX*w.d* format is similar to the DOLLARX*w.d* format, except that DOLLARX*w.d* format writes a leading dollar sign instead of the euro symbol.

Note: The EUROX*w.d* format uses the euro character (U+20AC). If you use the DBCS version of SAS and an encoding that does not support the euro character, an

error will occur. To prevent this error, change your session encoding to an encoding that supports the euro character.

Example

These examples use 1254.71 as the value of amount.

Statements	Results
`select put(1254.71,eurox10.2);`	`E1.254,71`
`select put(1254.71,eurox5.);`	`1.255`
`select put(1254.71,eurox9.2);`	`E1.254,71`
`select put(1254.71,eurox15.3);`	`E1.254,710`

See Also

Formats:

- "EUROw.d Format" on page 138

FLOATw.d Format

Generates a native single-precision, floating-point value by multiplying a number by 10 raised to the *d*th power.

Categories: Numeric
CAS

Alignment: Left

Syntax

FLOAT*w*.[*d*]

Arguments

w
specifies the width of the output field.

Requirement width must be 4

d

specifies the power of 10 by which to multiply the value.

Default 0

Range 0–31

Details

This format is useful in operating environments where a float value is not the same as a truncated double. Values that are written by FLOAT4. typically are those meant to be read by some other external program that runs in your operating environment and that expects these single-precision values. If the value that is to be formatted is a missing value, or if it is out-of-range for a native single-precision, floating-point value, a single-precision value of zero is generated.

Example

The following example illustrates the FLOAT*w.d* format:

Statements	Results[1]
`select put(1,float4.);`	`0000803F`

1 The result is a hexadecimal representation of a binary number that is stored in IEEE form.

See Also

Formats:

- "IEEEw.d Format" on page 148

FRACTw. Format

Converts numeric values to fractions.

Categories: Numeric
CAS

Alignment: Right

Syntax

FRACT*w.*

Arguments

w

specifies the width of the output field.

Default 10

Range 4–32

Details

Dividing the number 1 by 3 produces the value 0.33333333. To write this value as 1/3, use the FRACT*w*. format. FRACT*w*. writes fractions in reduced form, that is, 1/2 instead of 50/100.

Example

The following example illustrates the FRACT*w.* format:

Statements	Results
`select put(0.6666666667,fract8.);`	`2/3`
`select put(0.2784,fract8.);`	`174/625`

HEXw. Format

Converts real binary (floating-point) values to hexadecimal representation.

Categories: Numeric
CAS

Alignment: Left

Syntax

HEX*w.*

Arguments

w

specifies the width of the output field.

Default 8

Range 1–16

Tip If *w*< 16, the HEX*w*. format converts real binary numbers to fixed-point integers before writing them as hexadecimal characters. It also writes negative numbers in two's complement notation, and right aligns digits. If *w* is 16, HEX*w*. displays floating-point values in their hexadecimal form.

Details

In any operating environment, the least significant byte written by HEX*w*. is the rightmost byte. Some operating environments store integers with the least significant digit as the first byte. The HEX*w*. format produces consistent results in any operating environment regardless of the order of significance by byte.

Note: Different operating environments store floating-point values in different ways. However, the HEX16. format writes hexadecimal representations of floating-point values with consistent results in the same way that your operating environment stores them.

Comparisons

The HEX*w*. numeric format and the $HEX*w*. character format both generate the hexadecimal equivalent of values.

Example

The following examples illustrate the HEX*w.* format:

Statements	Results
`select put(35.4, hex8.);`	`00000023`
`select put(88, hex8.);`	`00000058`
`select put(2.33, hex8.);`	`00000002`

Statements	Results
`select put(-150, hex8.);`	`FFFFFF6A`

See Also

Formats:

- "BINARYw. Format" on page 110
- "$HEXw. Format" on page 98

HHMMw.d Format

Writes SAS time values as hours and minutes in the form *hh:mm*.

Categories: Date and Time
CAS

Alignment: Right

Syntax

HHMM*w*.[*d*]

Arguments

w

specifies the width of the output field.

Default 5

Range 2–20

d

specifies the number of digits to the right of the decimal point in the minutes value. The digits to the right of the decimal point specify a fraction of a minute.

Default 0

Range 0–19

Requirement must be less than *w*

Details

The HHMM*w.d* format writes SAS datetime values in the form *hh:mm*, where

hh
is an integer.

mm
is the number of minutes that range from 00 through 59.

SAS rounds hours and minutes that are based on the value of seconds in a SAS time value.

Comparisons

The HHMM*w.d* format is similar to the TIME*w.d* format except that the HHMM*w.d* format does not print seconds.

The HHMM*w.d* format and the TIME*w.d* format write a leading blank for the single-hour digit. The TOD*w.d* format writes a leading zero for a single-hour digit.

Example

The example table uses the input value of 46796, which is the SAS time value that corresponds to 12:59:56 PM.

Statements	Results
`select put(46796,hhmm.);`	`13:00`
`select put(46796,hhmm8.2);`	`12:59.93`

In the first example, SAS rounds up the time value four seconds based on the value of seconds in the SAS time value. In the second example, by adding a decimal specification of 2 to the format shows that fifty-six seconds is 93% of a minute.

See Also

Formats:

- “HOURw.d Format” on page 147
- “MMSSw.d Format” on page 154
- “TIMEw.d Format” on page 173
- “TODw.d Format” on page 177

Functions:

- "HOUR Function" on page 549
- "MINUTE Function" on page 649
- "SECOND Function" on page 806

HOURw.d Format

Writes SAS time values as hours and decimal fractions of hours.

Categories: Date and Time
CAS

Alignment: Right

Syntax

HOUR*w*.[*d*]

Arguments

w
specifies the width of the output field.

Default 2

Range 2–20

d
specifies the number of digits to the right of the decimal point in the hour value. Therefore, SAS prints decimal fractions of the hour.

Range 0–19

Requirement must be less than *w*

Details

SAS rounds hours based on the value of minutes in the SAS time value.

Example

The example table uses the input value of 41400, which is the SAS time value that corresponds to 11:30 AM.

Statements	Results
`select put(41400,hour4.1);`	`11.5`

See Also

Formats:

- “HHMMw.d Format” on page 145
- “MMSSw.d Format” on page 154
- “TIMEw.d Format” on page 173
- “TODw.d Format” on page 177

IEEEw.d Format

Generates an IEEE floating-point value by multiplying a number by 10 raised to the *d*th power.

Categories: Numeric
CAS

Alignment: Left

CAUTION: **Large floating-point values and floating-point values that require precision might not be identical to the original SAS value when they are written to an IBM mainframe by using the IEEE format and read back into SAS using the IEEE informat.**

Syntax

IEEE*w*.[*d*]

Arguments

w

specifies the width of the output field.

Default 8

Range 3–8

Tip If *w* is 8, an IEEE double-precision, floating-point number is written. If *w* is 5, 6, or 7, an IEEE double-precision, floating-point number is written, which assumes truncation of the appropriate number of bytes. If *w* is 4, an IEEE single-precision floating-point number is written. If *w* is 3, an

IEEE single-precision, floating-point number is written, which assumes truncation of one byte.

d

specifies to multiply the number by 10^d.

Default 0

Range 0–10

Details

This format is useful in operating environments where IEEE*w.d* is the floating-point representation that is used. In addition, you can use the IEEE*w.d* format to create files that are used by programs in operating environments that use the IEEE floating-point representation.

Typically, programs generate IEEE values in single-precision (4 bytes) or double-precision (8 bytes). Programs perform truncation solely to save space on output files. Machine instructions require that the floating-point number be one of the two lengths. The IEEE*w.d* format allows other lengths, which enables you to write data to files that contain space-saving truncated data.

Example

The following examples illustrate the IEEE*w.d* format:

Statements	Results[1]
`select put(1,ieee4.);`	`3FF00000`
`select put(1,ieee5.);`	`3FF0000000`

1 The result contains hexadecimal representations of binary numbers stored in IEEE form.

See Also

Formats:

- “FLOATw.d Format” on page 141

JULIANw. Format

Writes SAS date values as Julian dates in the form *yyddd* or *yyyyddd*.

Categories: Date and Time
CAS

Alignment: Left

Syntax

JULIAN*w.*

Arguments

w
specifies the width of the output field.

Default 5

Range 5–7

Tip If *w* is 5, the JULIAN*w*. format writes the date with a two-digit year. If *w* is 7, the JULIAN*w*. format writes the date with a four-digit year.

Details

The JULIAN*w*. format writes SAS date values in the form *yyddd* or *yyyyddd*, where

yy or *yyyy*
is a two-digit or four-digit integer that represents the year.

ddd
is the number of the day, 1–365 (or 1–366 for leap years), in that year.

Example

The example table uses the input value of 19431, which is the SAS date value that corresponds to March 14, 2013 (the 73rd day of the year).

Statements	Results
`select put(19431,julian5.);`	`13073`
`select put(19431,julian7.);`	`2013073`

See Also

Formats:

- "DATEw. Format" on page 115

MMDDYYw. Format

Writes SAS date values in the form *mmdd*[*yy*]*yy* or *mm*/*dd*/[*yy*]*yy*, where a forward slash is the separator and the year appears as either 2 or 4 digits.

Categories: Date and Time
CAS

Alignment: Right

Syntax

MMDDYY*w.*

Arguments

w
specifies the width of the output field.

Default 8

Range 2–10

Interaction When *w* has a value of from 2 to 5, the date appears with as much of the month and the day as possible. When *w* is 7, the date appears as a two-digit year without slashes.

Details

The MMDDYY*w*. format writes SAS date values in the form *mmdd*[*yy*]*yy* or *mm*/*dd*/[*yy*]*yy*, where

mm
is an integer that represents the month.

/
is the separator.

dd
is an integer that represents the day of the month.

[*yy*]*yy*
is a two-digit or four-digit integer that represents the year.

Example

The following examples use the input value of 19431, which is the SAS date value that corresponds to March 14, 2013.

Statements	Results
`select put(19431,mmddyy2.);`	`03`
`select put(19431,mmddyy3.);`	`03`
`select put(19431,mmddyy4.);`	`0314`
`select put(19431,mmddyy5.);`	`03/14`
`select put(19431,mmddyy6.);`	`031413`
`select put(19431,mmddyy7.);`	`031413`
`select put(19431,mmddyy8.);`	`03/14/13`
`select put(19431,mmddyy10.);`	`03/14/2013`

See Also

Formats:

- "DATEw. Format" on page 115
- "DDMMYYw. Format" on page 122
- "MMDDYYxw. Format" on page 152
- "YYMMDDw. Format" on page 191

MMDDYYxw. Format

Writes SAS date values in the form *mmdd*[*yy*]*yy* or *mm*X*dd*X[*yy*]*yy*, where X represents a specified separator and the year appears as either 2 or 4 digits.

Categories: Date and Time
CAS

Alignment: Right

Syntax

MMDDYY*xw.*

Arguments

x

identifies a separator or specifies that no separator appear between the month, the day, and the year. Valid values for *x* are any of the following:

B
separates with a blank

C
separates with a colon

D
separates with a hyphen

N
indicates no separator

P
separates with a period

S
separates with a slash.

w

specifies the width of the output field.

Default 8

Range 2–10

Interactions When *w* has a value of from 2 to 5, the date appears with as much of the month and the day as possible. When *w* is 7, the date appears as a two-digit year without separators.

When *x* has a value of N, the width range changes to 2–8.

Details

The MMDDYY*xw*. format writes SAS date values in the form *mmdd*[*yy*]*yy* or *mm*X*dd*X[*yy*]*yy*, where

mm
is an integer that represents the month.

X
is a specified separator.

dd
is an integer that represents the day of the month.

[*yy*]*yy*
is a two-digit or four-digit integer that represents the year.

Example

The following examples use the input value of 19431, which is the SAS date value that corresponds to March, 14, 2013.

Statements	Results
`select put(19431,mmddyyc5.);`	`03:14`
`select put(19431,mmddyyd8.);`	`03-14-13`
`select put(19431,mmddyyn9.);`	`03132013`
`select put(19431,mmddyyp10.);`	`03.14.2013`

See Also

Formats:

- “DATEw. Format” on page 115
- “DDMMYYxw. Format” on page 124
- “MMDDYYw. Format” on page 151
- “YYMMDDxw. Format” on page 193

MMSSw.d Format

Writes SAS time values as the number of minutes and seconds since midnight.

Categories: Date and Time
CAS

Alignment: Right

Syntax

MMSS*w*.[*d*]

Arguments

w

specifies the width of the output field.

Default 5

Range 2–20

Tip Set *w* to a minimum of 5 to write a value that represents minutes and seconds.

d

specifies the number of digits to the right of the decimal point in the seconds value. Therefore, the SAS time value includes fractional seconds.

Range 0–19

Restriction must be less than *w*

Example

The example table uses the SAS input value of 4530.

Statements	Results
`select put(4530,mmss.);`	`75:30`

See Also

Formats:

- "HHMMw.d Format" on page 145
- "TIMEw.d Format" on page 173

MMYYw. Format

Writes SAS date values in the form *mm*M[*yy*]*yy*, where M is the separator and the year appears as either 2 or 4 digits.

Categories: Date and Time
CAS

Alignment: Right

Syntax

MMYY*w.*

Arguments

w

specifies the width of the output field.

Default 7

Range 5–32

Interaction When *w* has a value of 5 or 6, the date appears with only the last two digits of the year. When *w* is 7 or more, the date appears with a four-digit year.

Details

The MMYY*w*. format writes SAS date values in the form *mm*M<*yy*>*yy*, where

mm

is an integer that represents the month.

M

is the character separator.

[*yy*]*yy*

is a two-digit or four-digit integer that represents the year.

Example

The following examples use the input value of 19431, which is the SAS date value that corresponds to March 14, 2013.

Statements	Results
`select put(19431,mmyy5.);`	`03M13`
`select put(19431,mmyy6.);`	`03M13`
`select put(19431,mmyy7.);`	`03M2013`
`select put(19431,mmyy10.);`	`03M2013`

See Also

Formats:

- “DATEw. Format” on page 115
- “MMYYxw. Format” on page 157

■ "YYMMw. Format" on page 187

MMYYxw. Format

Writes SAS date values in the form *mm*[*yy*]*yy* or *mm*X[*yy*]*yy*. The *x* in the format name is a character that represents the special character. The special character separates the month and the year. That special character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits.

Categories: Date and Time
CAS

Alignment: Right

Syntax

MMYY*xw.*

Arguments

x

identifies a separator or specifies that no separator appear between the month and the year. Valid values for *x* are any of the following:

C
separates with a colon

D
separates with a hyphen

N
indicates no separator

P
separates with a period

S
separates with a forward slash.

w

specifies the width of the output field.

Default 7

Range 5–32

Interactions When *x* is set to N, no separator is specified. The width range is then 4–32, and the default changes to 6.

When *x* has a value of C, D, P, or S and *w* has a value of 5 or 6, the date appears with only the last two digits of the year. When *w* is 7 or more, the date appears with a four-digit year.

When *x* has a value of *N* and *w* has a value of 4 or 5, the date appears with only the last two digits of the year. When *x* has a value of *N* and *w* is 6 or more, the date appears with a four-digit year.

Details

The MMYY*xw*. format writes SAS date values in the form *mm*[*yy*]*yy* or *mm*X[*yy*]*yy*, where

mm
is an integer that represents the month.

X
is a specified separator.

[*yy*]*yy*
is a two-digit or four-digit integer that represents the year.

Example

The following examples use the input value of 19560, which is the SAS date value that corresponds to July14, 2013.

Statements	Results
`select put(19560,mmyyc5.);`	07:13
`select put(19560,mmyyd7.);`	07-2013
`select put(19560,mmyyn4.);`	0713
`select put(19560,mmyyp8.);`	07.2013
`select put(19560,mmyys10.);`	07/2013

See Also

Formats:

- “DATEw. Format” on page 115
- “MMYYw. Format” on page 155
- “YYMMw. Format” on page 187

MONNAMEw. Format

Writes SAS date values as the name of the month.

Categories: Date and Time
CAS

Alignment: Right

Syntax

MONNAME*w.*

Arguments

w
specifies the width of the output field.

Default 9

Range 1–32

Tip Use MONNAME3. to print the first three letters of the month name.

Details

If necessary, SAS truncates the name of the month to fit the format width.

Example

The example table uses the input value of 19431, which is the SAS date value that corresponds to March 14, 2014.

Statements	Results
`select put(19431,monname1.);`	`M`
`select put(19431,monname3.);`	`Mar`
`select put(19431,monname5.);`	`March`

See Also

Formats:

- "MONTHw. Format" on page 160

MONTHw. Format

Writes SAS date values as the month of the year.

Categories: Date and Time
CAS

Alignment: Right

Syntax

MONTH*w.*

Arguments

w

specifies the width of the output field.

Default 2

Range 1–32

Details

The MONTH*w*. format writes the month (1 through 12) of the year from a SAS date value.

Example

The example table uses the input value of 19431, which is the SAS date value that corresponds to March 14, 2014.

Statements	Results
`select put(19431,month.);`	3

See Also

Formats:

- "MONNAMEw. Format" on page 159

MONYYw. Format

Writes SAS date values as the month and the year in the form *mmmyy* or *mmmyyyy*.

Categories: Date and Time
CAS

Alignment: Right

Syntax

MONYY*w.*

Arguments

w
specifies the width of the output field.

Default 5

Range 5–7

Details

The MONYY*w*. format writes SAS date values in the form *mmmyy* or *mmmyyyy*, where

mmm
is the first three letters of the month name.

yy or *yyyy*
is a two-digit or four-digit integer that represents the year.

Comparisons

The MONYY*w*. format and the DTMONYY*w*. format are similar in that they both write date values. The difference is that MONYY*w*. expects a SAS date value as input, and DTMONYY*w*. expects a datetime value.

Example

The example table uses the input value of 19704, which is the SAS date value that corresponds to December 12, 2013.

Statements	Results
`select put(19704,monyy5.);`	DEC13
`select put(19704,monyy7.);`	DEC2013

See Also

Formats:

- “DATEw. Format” on page 115
- “DTMONYYw. Format” on page 131
- “DDMMYYw. Format” on page 122
- “MMDDYYw. Format” on page 151
- “YYMMDDw. Format” on page 191

Functions:

- “MONTH Function” on page 655
- “YEAR Function” on page 864

NEGPARENw.d Format

Writes negative numeric values in parentheses.

Categories: Numeric
CAS

Alignment: Right

Syntax

NEGPAREN*w*.[*d*]

Arguments

w

specifies the width of the output field.

Default 6

Range 1–32

d

specifies the number of digits to the right of the decimal point in the numeric value.

Default 0

Range 0–31

Details

The NEGPAREN*w.d* format attempts to right align output values. If the input value is negative, NEGPAREN*w.d* displays the output by enclosing the value in parentheses, if the field that you specify is wide enough. Otherwise, it uses a minus sign to represent the negative value. If the input value is non-negative, NEGPAREN*w.d* displays the value with a leading and trailing blank to ensure proper column alignment. It reserves the last column for a close parenthesis even when the value is positive.

Comparisons

The NEGPAREN*w.d* format is similar to the COMMA*w.d* format in that it separates every three digits of the value with a comma.

Example

The following examples illustrate the NEGPAREN*w.d* format:

Statements	Results
`select put(100,negparen6.);`	`100`
`select put(-200,negparen6.);`	`(200)`
`select put(-2000,negparen8.);`	`(2,000)`

NENGOw. Format

Writes SAS date values as Japanese dates in the form *e.yymmdd*.

Categories: Date and Time
CAS

Alignment: Left

Syntax

NENGO*w.*

Arguments

w
specifies the width of the output field.

Default 10

Range 2–10

Details

The NENGO*w*. format writes SAS date values in the form *e.yymmdd*, where

e
is the first letter of the name of the imperial era (Meiji, Taisho, Showa, Heisei, or Reiwa).

yy
is an integer that represents the year.

mm
is an integer that represents the month.

dd
is an integer that represents the day of the month.

If the width is too small, SAS omits the period.

Example

The example table uses the input value of 19431, which is the SAS date value that corresponds to March 14, 2013.

Statements	Results
`select put(19431,nengo3.);`	H25
`select put(19431,nengo6.);`	H25/03
`select put(19431,nengo8.);`	H.250314
`select put(19431,nengo9.);`	H25/03/14
`select put(19431,nengo10.);`	H.25/03/14

See Also

Formats:

- "DATEw. Format" on page 115

OCTALw. Format

Converts numeric values to octal representation.

Categories: Numeric
CAS

Alignment: Left

Syntax

OCTAL*w.*

Arguments

w

specifies the width of the output field.

Default 3

Range 1–24

Details

If necessary, the OCTAL*w*. format converts numeric values to integers before displaying them in octal representation.

Comparisons

OCTAL*w*. converts numeric values to octal representation. The $OCTAL*w*. format converts character values to octal representation.

Example

The following example illustrates the OCTAL*w.* format:

Statements	Results
`select put (3592, octal16.);`	`0000000000007010`

See Also

Formats:

- "$OCTALw. Format" on page 99

PERCENTw.d Format

Writes numeric values as percentages.

Categories: Numeric
CAS

Alignment: Right

Syntax

PERCENT*w*.[*d*]

Arguments

w

specifies the width of the output field.

Default 6

Range 4–32

d

specifies the number of digits to the right of the decimal point in the numeric value.

Range 0–31

Requirement must be less than *w*

Details

The PERCENT*w.d* format multiplies values by 100, formats them the same as the BEST*w.d* format, and adds a percent sign (%) to the end of the formatted value, while it encloses negative values in parentheses. The PERCENT*w.d* format allows room for a percent sign and parentheses, even if the value is not negative.

Example

The following examples illustrate the PERCENT*w.d* format:

Statements	Results
`select put(0.1,percent10.);`	`10%`
`select put(1.2,percent10.);`	`120%`
`select put(-.05,percent10.);`	`(5%)`

See Also

Formats:

- "PERCENTNw.d Format" on page 168

PERCENTNw.d Format

Produces percentages, using a minus sign for negative values.

Categories: Numeric
CAS

Alignment: Right

Syntax

PERCENTN*w*.[*d*]

Arguments

w

specifies the width of the output field.

Default 6

Range 4–32

d

specifies the number of digits to the right of the decimal point in the numeric value.

Range 0–31

Requirement must be less than *w*

Details

The PERCENTN*w.d* format multiplies negative values by 100, formats them the same as the BEST*w.d* format, adds a minus sign to the beginning of the value, and adds a percent sign (%) to the end of the formatted value. The PERCENTN*w.d* format allows room for a percent sign and a minus sign, even if the value is not negative.

Comparisons

The PERCENTN*w.d* format produces percents by using a minus sign instead of parentheses for negative values. The PERCENT*w.d* format produces percents by using parentheses for negative values.

Example

The following examples illustrate the PERCENTN*w.d* format:

Statements	Results
`select put(-0.1,percentn10.);`	`-10%`
`select put(.2,percentn10.);`	`20%`
`select put(.8,percentn10.);`	`80%`
`select put(-0.05,percentn10.);`	`-5%`
`select put(-6.3,percentn10.);`	`-630%`

See Also

Formats:

- "PERCENTw.d Format" on page 166

QTRw. Format

Writes SAS date values as the quarter of the year.

Categories: Date and Time
CAS

Alignment: Right

Syntax

QTR*w.*

Arguments

w

specifies the width of the output field.

Default 1

Range 1–32

Example

The example table uses the input value of 19624, which is the SAS date value that corresponds to September 23, 2013.

Statements	Results
`select put(19624,qtr.);`	3

See Also

Formats:

- “DATEw. Format” on page 115
- “QTRRw. Format” on page 170
- “YYQw. Format” on page 196
- “YYQxw. Format” on page 197
- “YYQZw. Format” on page 202

QTRRw. Format

Writes SAS date values as the quarter of the year in Roman numerals.

Categories: Date and Time
CAS

Alignment: Right

Syntax

QTRR*w.*

Arguments

w

specifies the width of the output field.

Default 3

Range 3–32

Example

The example table uses the input value of 19624, which is the SAS date value that corresponds to September 23, 2013.

Statements	Results
`select put(19624,qtrr.);`	`III`

See Also

Formats:

- "QTRw. Format" on page 169
- "YYQRw. Format" on page 199
- "YYQRxw. Format" on page 201

ROMANw. Format

Writes numeric values as roman numerals.

Categories: Numeric
CAS

Alignment: Left

Syntax

ROMAN*w.*

Arguments

w

specifies the width of the output field.

Default 6

Range 2–32

Details

The ROMAN*w*. format truncates a floating-point value to its integer component before the value is written.

Example

The following example illustrates the ROMAN*w.* format:

Statements	Results
`select put(2006,roman.);`	`MMVI`

See Also

Formats:

- "DATEw. Format" on page 115
- "JULIANw. Format" on page 149

SIZEKw.d Format

Writes a numeric value in the form *n*K for kilobytes.

Categories: Numeric
CAS

Alignment: Right

Syntax

SIZEK*w*.[*d*]

Arguments

w

specifies the width of the output field.

Default 9

Range 2–33

d

specifies the number of digits to the right of the decimal point in the numeric value.

Default 0

Range 0–31

Details

To write a numeric value in the form *n*K by using the SIZEK*w.d* format, the value of *n* is calculated by dividing the numeric value by 1,024. The symbol K indicates that the value is a multiple of 1,024.

Example

The following examples illustrate the SIZE*w.d* format:

Statements	Results
`select put (1024, sizek.);`	`1K`
`select put (200943, sizek.);`	`197K`

TIMEw.d Format

Writes SAS time values as hours, minutes, and seconds in the form *hh:mm:ss.ss* using the military 24-hour clock.

Categories: Date and Time
CAS

Alignment: Right

Syntax

TIME*w*.[*d*]

Arguments

w

specifies the width of the output field.

Default 8

Range 2–20

Tip Make *w* large enough to produce the desired results. To obtain a complete time value with three decimal places, you must allow at least 12 spaces: eight spaces to the left of the decimal point, one space for the decimal point itself, and three spaces for the decimal fraction of seconds.

d
specifies the number of digits to the right of the decimal point in the seconds value.

Default 0

Range 0–19

Requirement must be less than *w*

Details

The TIME*w.d* format writes SAS time values in the form *hh:mm:ss.ss*, where

hh
is an integer.

Note: If *hh* is a single digit, TIME*w.d* places a leading blank before the digit. For example, the TIME*w.d.* format writes 9:00 instead of 09:00.

mm
is the number of minutes, ranging from 00 through 59.

ss.ss
is the number of seconds, ranging from 00 through 59, with the fraction of a second following the decimal point.

Comparisons

The TIME*w.d* format is similar to the HHMM*w.d* format except that TIME*w.d* includes seconds.

The TIME*w.d* format and the HHMM*w* format write a leading blank for a single-hour digit. The TOD*w.d* format writes a leading zero for a single-hour digit.

Example

The example table uses the input value of 59083, which is the SAS time value that corresponds to 4:24:43 PM.

Statements	Results
`select put(59083,time.);`	`16:24:43`

See Also

Formats:

- “HHMMw.d Format” on page 145
- “HOURw.d Format” on page 147
- “MMSSw.d Format” on page 154
- “TODw.d Format” on page 177

Functions:

- “HOUR Function” on page 549
- “MINUTE Function” on page 649
- “SECOND Function” on page 806

TIMEAMPMw.d Format

Writes SAS time values as hours, minutes, and seconds in the form *hh:mm:ss.ss* with AM or PM.

Categories: Date and Time
CAS

Alignment: Right

Syntax

TIMEAMPM*w*.[*d*]

Arguments

w

specifies the width of the output field.

Default 11

Range 2–20

d

specifies the number of digits to the right of the decimal point in the seconds value.

Default 0

Range 0–19

Requirement must be less than *w*

Details

The TIMEAMPM*w.d* format writes SAS time values in the form *hh:mm:ss.ss* with AM or PM, where

hh
is an integer that represents the hour.

mm
is an integer that represents the minutes.

ss.ss
is the number of seconds to two decimal places.

Times greater than 23:59:59 PM appear as the next day.

Make *w* large enough to produce the desired results. To obtain a complete time value with three decimal places and AM or PM, you must allow at least 11 spaces (*hh:mm:ss* PM). If *w* is less than 5, SAS writes AM or PM only.

Comparisons

- The TIMEAMPMM*w.d* format is similar to the TIMEM*w.d* format except, that TIMEAMPMM*w.d* prints AM or PM at the end of the time.
- TIME*w.d* writes hours greater than 11:59:59, and TIMEAMPM*w.d* does not.

Example

The example table uses the input value of 59083, which is the SAS time value that corresponds to 4:24:43 PM.

Statements	Results
`select put(59083,timeampm3.);`	`PM`
`select put(59083,timeampm5.);`	`4 PM`
`select put(59083,timeampm7.);`	`4:24 PM`
`select put(59083,timeampm11.);`	`4:24:43 PM`

See Also

Formats:

- "DATEAMPMw.d Format" on page 117
- "TIMEw.d Format" on page 173

TODw.d Format

Writes SAS time values and the time portion of SAS datetime values in the form *hh:mm:ss.ss*.

Categories: Date and Time
CAS

Alignment: Right

Syntax

TOD*w*.[*d*]

Arguments

w

specifies the width of the output field.

Default 8

Range 2–20

Tip SAS writes a zero for a zero hour if the specified width is sufficient, for example, 02:30 or 00:30.

d

specifies the number of digits to the right of the decimal point in the seconds value.

Default 0

Range 0–19

Requirement must be less than *w*

Details

The TOD*w.d* format writes SAS datetime values in the form *hh:mm:ss.ss*, where

hh
is an integer that represents the hour.

mm
is an integer that represents the minutes.

ss.ss
is the number of seconds to two decimal places.

Comparisons

The TOWw.d format writes a leading zero for a single-hour digit. The TIMEw.d format and the HHMMw.d format write a leading blank for a single-hour digit.

Example

The following example illustrates the TOD*w.d* format:

Statements	Results
`select put(1472049623,tod9.);`	`14:40:23`

See Also

Formats:

- "TIMEw.d Format" on page 173
- "TIMEAMPMw.d Format" on page 175

VAXRBw.d Format

Writes real binary (floating-point) data in VMS format.

Categories: Numeric
CAS

Alignment: Right

Syntax

VAXRB*w*.[*d*]

Arguments

w

specifies the width of the output field.

Default 8

Range 2–8

d

specifies the power of 10 by which to divide the value.

Default 0

Range 0–31

Details

Use the VAXRB*w.d* format to write data in native VAX/VMS floating-point notation.

Example

The following example illustrates the VAXRB*w.d* format:

Statements	Results[1]
`select put(1,vaxrb8.);`	`8040000000000000`

1 The result is the hexadecimal representation for the integer.

w.d Format

Writes standard numeric data one digit per byte.

Categories: Numeric
CAS

Alignment: Right

Alias: F*w.d*

Syntax

w.[*d*]

Arguments

w

specifies the width of the output field.

Range 1–32

Tip Allow enough space to write the value, the decimal point, and a minus sign, if necessary.

d

specifies the number of digits to the right of the decimal point in the numeric value. This argument is optional.

Range 0–31

Requirement must be less than *w*

Tip If *d* is 0 or you omit *d*, *w.d* writes the value without a decimal point.

Details

The *w.d* format rounds to the nearest number that fits in the output field. If *w.d* is too small, SAS might shift the decimal to the BEST*w*. format. The *w.d* format writes negative numbers with leading minus signs. In addition, *w.d* right aligns before writing and pads the output with leading blanks.

Comparisons

The Z*w.d* format is similar to the *w.d* format except that Z*w.d* pads right-aligned output with 0s instead of blanks.

Example

The following example illustrates the *w.d* format:

Statements	Results
`select put (23.45, 6.3);`	`23.450`

See Also

Formats:

- "BESTw. Format" on page 106
- "BESTDw.p Format" on page 108
- "Dw.p Format" on page 114
- "$w. Format" on page 105
- "Zw.d Format" on page 204

WEEKDATEw. Format

Writes SAS date values as the day of the week and the date in the form *day-of-week*, *month-name dd*, *yy* (or *yyyy*).

Categories: Date and Time
CAS

Alignment: Right

Syntax

WEEKDATE*w.*

Arguments

w
specifies the width of the output field.

Default 29

Range 3–37

Details

The WEEKDATE*w*. format writes SAS date values in the form *day-of-week*, *month-name dd*, *yy* (or *yyyy*), where

dd
is an integer that represents the day of the month.

yy or *yyyy*
is a two-digit or four-digit integer that represents the year.

If *w* is too small to write the complete day of the week and month, SAS abbreviates as needed.

Comparisons

The WEEKDATE*w*. format is the same as the WEEKDATX*w*. format except that WEEKDATX*w*. prints *dd* before the month's name.

Example

The example table uses the input value of 19537 which is the SAS date value that corresponds to June 28, 2013.

Statements	Results
`select put(19537,weekdate3.);`	`Fri`
`select put(19537,weekdate9.);`	`Friday`
`select put(19537,weekdate15.);`	`Fri, Jun 28, 13`
`select put(19537,weekdate17.);`	`Fri, Jun 28, 2013`

See Also

Formats:

- “DTWKDATXw. Format” on page 133
- “DATEw. Format” on page 115
- “DDMMYYw. Format” on page 122
- “MMDDYYw. Format” on page 151
- “TODw.d Format” on page 177
- “WEEKDATXw. Format” on page 182
- “YYMMDDw. Format” on page 191

WEEKDATXw. Format

Writes SAS date values as the day of the week and date in the form *day-of-week*, *dd month-name yy* (or *yyyy*).

Categories: Date and Time
CAS

Alignment: Right

Syntax

WEEKDATX*w.*

Arguments

w
specifies the width of the output field.

Default 29

Range 3–37

Details

The WEEKDATX*w*. format writes SAS date values in the form *day-of-week*, *dd month-name*, *yy* (or *yyyy*), where

dd
is an integer that represents the day of the month.

yy or *yyyy*
is a two-digit or a four-digit integer that represents the year.

If *w* is too small to write the complete day of the week and month, then SAS abbreviates as needed.

Comparisons

The WEEKDATE*w*. format is the same as the WEEKDATX*w*. format, except that WEEKDATE*w*. prints *dd* after the month's name.

The WEEKDATX*w*. format is the same as the DTWKDATX*w*. format, except that DTWKDATX*w*. expects a datetime value as input.

Example

The example table uses the input value of 19405, which is the SAS date value that corresponds to February 16, 2013.

Statements	Results
`select put(19405,weekdatx.);`	`Saturday, 16 February 2013`

See Also

Formats:

- "DTWKDATXw. Format" on page 133
- "DATEw. Format" on page 115
- "DDMMYYw. Format" on page 122
- "MMDDYYw. Format" on page 151
- "TODw.d Format" on page 177
- "WEEKDATEw. Format" on page 181
- "YYMMDDw. Format" on page 191

WEEKDAYw. Format

Writes SAS date values as the day of the week.

Categories:	Date and Time CAS
Alignment:	Right

Syntax

WEEKDAY*w.*

Arguments

w

specifies the width of the output field.

Default 1

Range 1–32

Details

The WEEKDAY*w*. format writes a SAS date value as the day of the week (where 1=Sunday, 2=Monday, and so on).

Example

The example table uses the input value of 19405, which is the SAS date value that corresponds to February 16, 2013.

Statements	Results
`select put(19405,weekday.);`	7

See Also

Formats:

- "DOWNAMEw. Format" on page 129

YEARw. Format

Writes SAS date values as the year.

Categories: Date and Time
CAS

Alignment: Right

Syntax

YEAR*w.*

Arguments

w

specifies the width of the output field.

Default 4

Range 2–32

Tip If *w* is less than 4, the last two digits of the year print. Otherwise, the year value prints as four digits.

Comparisons

The YEAR*w*. format is similar to the DTYEAR*w*. format in that they both write date values. The difference is that YEAR*w*. expects a SAS date value as input, and DTYEAR*w*. expects a SAS datetime value.

Example

The example table uses the input value of 19537, which is the SAS date value that corresponds to June 28, 2013.

Statements	Results
`select put(19537,year2.);`	13
`select put(19537,year4.);`	2013

See Also

Formats:

- "DTYEARw. Format" on page 134

YENw.d Format

Writes numeric values with yen signs, commas, and decimal points.

Categories: Numeric
CAS

Alignment: Right

Syntax

YEN*w.d*

Arguments

w

specifies the width of the output field.

Default 1

Range 1–32

d

specifies the number of digits to the right of the decimal point in the numeric value.

Restriction must be either 0 or 2

Tip If *d* is 2, then YEN*w.d* writes a decimal point and two decimal digits. If *d* is 0, then YEN*w.d* does not write a decimal point or decimal digits.

Details

The YEN*w.d* format writes numeric values with a leading yen sign and with a comma that separates every three digits of each value.

The hexadecimal representation of the code for the yen sign character is 5B on EBCDIC systems and 5C on ASCII systems. The monetary character these codes represent can be different in other countries.

Example

The following example illustrates the YEN*w.d* format:

Statements	Results
`select put(1254.71,yen10.2);`	`¥1,254.71`

See Also

Formats:

- “DOLLARw.d Format” on page 126
- “EUROw.d Format” on page 138

YYMMw. Format

Writes SAS date values in the form [*yy*]*yy*M*mm*, where M is the separator and the year appears as either 2 or 4 digits.

Categories: Date and Time
CAS

Alignment: Right

Syntax

YYMM*w.*

Arguments

w

specifies the width of the output field.

Default 7

Range 5–32

Interaction When *w* has a value of 5 or 6, the date appears with only the last two digits of the year. When *w* is 7 or more, the date appears with a four-digit year.

Details

The YYMM*w*. format writes SAS date values in the form [*yy*]*yy*M*mm*, where

[*yy*]*yy*

is a two-digit or four-digit integer that represents the year.

M

is the character separator.

mm

is an integer that represents the month.

Example

The following examples use the input value of 19656, which is the SAS date value that corresponds to October 25, 2013.

Statements	Results
`select put(19656,yymm5.);`	`13M10`
`select put(19656,yymm6.);`	`13M10`
`select put(19656,yymm.);`	`2013M10`
`select put(19656,yymm7.);`	`2013M10`

Statements	Results
`select put(19656,yymm10.);`	`2013M10`

See Also

Formats:

- “DATEw. Format” on page 115
- “MMYYw. Format” on page 155
- “YYMMxw. Format” on page 189

YYMMxw. Format

Writes SAS date values in the form [*yy*]*yymm* or [*yy*]*yy-mm*. The *x* in the format name represents the special character that separates the year and the month. This special character can be a hyphen (-), period (.), slash(/), colon(:), or no separator. The year can be either 2 or 4 digits.

Categories: Date and Time
CAS

Alignment: Right

Syntax

YYMM*xw.*

Arguments

x

identifies a separator or specifies that no separator appear between the year and the month. Valid values for *x* are any of the following:

C
separates with a colon

D
separates with a hyphen

N
indicates no separator

P
separates with a period

S
separates with a forward slash

w
specifies the width of the output field.

Default 7

Range 5–32

Interactions When *x* is set to *N*, no separator is specified. The width range is then 4–32, and the default changes to 6.

When *x* has a value of C, D, P, or S and *w* has a value of 5 or 6, the date appears with only the last two digits of the year. When *w* is 7 or more, the date appears with a four-digit year.

When *x* has a value of *N* and *w* has a value of 4 or 5, the date appears with only the last two digits of the year. When *x* has a value of *N* and *w* is 6 or more, the date appears with a four-digit year.

Details

The YYMM*xw*. format writes SAS date values in one of the following forms:

yymmdd
[*yy*]*yy-mmxdd*

<yy>yy
is a two-digit or four-digit integer that represents the year.

x
is a specified separator.

mm
is an integer that represents the month.

Example

The following examples use the input value of 19537, which is the SAS date value that corresponds to June 28, 2013.

Statements	Results
`select put(19537,yymmc5.);`	13:06
`select put(19537,yymmd.);`	2013-06
`select put(19537,yymmn4.);`	1306
`select put(19537,yymmp8.);`	2013.06
`select put(19537,yymms10.);`	2013/06

See Also

Formats:

- "DATEw. Format" on page 115
- "MMYYxw. Format" on page 157
- "YYMMw. Format" on page 187

YYMMDDw. Format

Writes SAS date values in the form *yymmdd* or [*yy*]*yy-mm-dd*, where a hyphen (-) is the separator and the year appears as either 2 or 4 digits.

Categories: Date and Time
CAS

Alignment: Right

Syntax

YYMMDD*w.*

Arguments

w
specifies the width of the output field.

Default 8

Range 2–10

Interaction When *w* has a value of from 2 to 5, the date appears with as much of the year and the month as possible. When *w* is 7, the date appears as a two-digit year without a hyphen.

Details

The YYMMDD*w*. format writes SAS date values in one of the following forms:

yymmdd
[*yy*]*yy-mm-dd*

[*yy*]*yy*
is a two-digit or four-digit integer that represents the year.

-
is the separator.

mm
is an integer that represents the month.

dd
is an integer that represents the day of the month.

To format a date that has a four-digit year and no separators, use the YYMMDDx. format.

Example

The following examples use the input value of 19450, which is the SAS date value that corresponds to April 2, 2013.

Statements	Results
`select put(19450,yymmdd2.);`	13
`select put(19450,yymmdd3.);`	13
`select put(19450,yymmdd4.);`	1304
`select put(19450,yymmdd5.);`	13-04
`select put(19450,yymmdd6.);`	130402
`select put(19450,yymmdd7.);`	130402
`select put(19450,yymmdd8.);`	13-04-02
`select put(19450,yymmdd10.);`	2013-04-02

See Also

Formats:

- “DATEw. Format” on page 115
- “DDMMYYw. Format” on page 122
- “MMDDYYw. Format” on page 151
- “YYMMDDxw. Format” on page 193

Functions:

- “DAY Function” on page 420
- “MONTH Function” on page 655

■ "YEAR Function" on page 864

YYMMDDxw. Format

Writes date values in the form [*yy*]*yymmdd* or [*yy*]*yy-mm-dd*. The *x* in the format name is a character that represents the special character which separates the year, month, and day. This special character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits.

Categories: Date and Time
CAS

Alignment: Right

Syntax

YYMMDD*xw.*

Arguments

x

identifies a separator or specifies that no separator appear between the year, the month, and the day. Valid values for *x* are any of the following:

B
separates with a blank

C
separates with a colon

D
separates with a hyphen

N
indicates no separator

P
separates with a period

S
separates with a slash.

w

specifies the width of the output field.

Default 8

Range 2–10

Interactions When *w* has a value of from 2 to 5, the date appears with as much of the year and the month. When *w* is 7, the date appears as a two-digit year without separators.

When *x* has a value of N, the width range is 2–8.

Details

The YYMMDD*xw*. format writes SAS date values in one of the following forms:

yymmdd
[*yy*]*yyxmmxdd*

where

[*yy*]*yy*
is a two-digit or four-digit integer that represents the year.

x
is a specified separator.

mm
is an integer that represents the month.

dd
is an integer that represents the day of the month.

Example

The following examples use the input value of 19704, which is the SAS date value that corresponds to December 12, 2013.

Statements	Results
`select put(19704,yymmddc5.);`	`13:12`
`select put(19704,yymmddd8.);`	`13-12-12`
`select put(19704,yymmddp10.);`	`2013.12.12`

See Also

Formats:

- “DATEw. Format” on page 115
- “DDMMYYxw. Format” on page 124
- “MMDDYYxw. Format” on page 152
- “YYMMDDw. Format” on page 191

Functions:

- “DAY Function” on page 420
- “MONTH Function” on page 655
- “YEAR Function” on page 864

YYMONw. Format

Writes SAS date values in the form *yymmm* or *yyyymmm*.

Categories: Date and Time
CAS

Alignment: Right

Syntax

YYMON*w.*

Arguments

w

specifies the width of the output field. If the format width is too small to print a four-digit year, only the last two digits of the year are printed.

Default 7

Range 5–32

Details

The YYMON*w*. format abbreviates the month's name to three characters.

Example

The example table uses the input value of 19537, which is the SAS date value that corresponds to June 28,2013.

Statements	Results
`select put(19537,yymon6.);`	`13JUN`
`select put(19537,yymon7.);`	`2013JUN`

See Also

Formats:

- “DATEw. Format” on page 115
- “MMYYw. Format” on page 155

YYQw. Format

Writes SAS date values in the form [*yy*]*yy*Q*q*, where Q is the separator, the year appears as either 2 or 4 digits, and *q* is the quarter of the year.

Categories:	Date and Time CAS
Alignment:	Right

Syntax

YYQ*w.*

Arguments

w

specifies the width of the output field.

Default	6
Range	4–32
Interaction	When *w* has a value of 4 or 5, the date appears with only the last two digits of the year. When *w* is 6 or more, the date appears with a four-digit year.

Details

The YYQ*w*. format writes SAS date values in the form [*yy*]*yy*Q*q*, where

[*yy*]*yy*

is a two-digit or four-digit integer that represents the year.

Q

is the character separator.

q

is an integer (1, 2, 3, or 4) that represents the quarter of the year.

Example

The following examples use the input value of 19537, which is the SAS date value that corresponds to June 28, 2013.

Statements	Results
`select put(19537,yyq4.);`	13Q2
`select put(19537,yyq5.);`	13Q2
`select put(19537,yyq.);`	2013Q2
`select put(19537,yyq6.);`	2013Q2
`select put(19537,yyq10.);`	2013Q2

See Also

Formats:

- "DATEw. Format" on page 115
- "YYQxw. Format" on page 197
- "YYQRw. Format" on page 199
- "YYQRxw. Format" on page 201
- "YYQZw. Format" on page 202

YYQxw. Format

Writes SAS date values in the form [*yy*]*yyq* or [*yy*]*yy-q*. The *x* in the format name is a character that represents the special character that separates the year and the quarter of the year. This character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits.

Categories: Date and Time
CAS

Alignment: Right

Syntax

YYQ*xw.*

Arguments

x

identifies a separator or specifies that no separator appear between the year and the quarter. Valid values for *x* are any of the following:

C
separates with a colon

D
separates with a hyphen

N
indicates no separator

P
separates with a period

S
separates with a forward slash.

w

specifies the width of the output field.

Default 6

Range 4–32

Interactions When *x* is set to *N*, no separator is specified. The width range is then 3–32, and the default changes to 5.

When *w* has a value of 4 or 5, the date appears with only the last two digits of the year. When *w* is 6 or more, the date appears with a four-digit year.

When *x* has a value of *N* and *w* has a value of 3 or 4, the date appears with only the last two digits of the year. When *x* has a value of *N* and *w* is 5 or more, the date appears with a four-digit year.

Details

The YYQ*xw*. format writes SAS date values in one of the following forms:

[*yy*]*yyq*
[*yy*]*yyxq*

where

[*yy*]*yy*
is a two-digit or four-digit integer that represents the year.

X
is a specified separator.

q
is an integer (1, 2, 3, or 4) that represents the quarter of the year.

Example

The following examples use the input value of 19537, which is the SAS date value that corresponds to July 28, 2013.

Statements	Results
`select put(19537,yyqc4.);`	`13:2`
`select put(19537,yyqd.);`	`2013-2`
`select put(19537,yyqn3.);`	`132`
`select put(19537,yyqp6.);`	`2013.2`
`select put(19537,yyqs8.);`	`2013/2`

See Also

Formats:

- “DATEw. Format” on page 115
- “YYQw. Format” on page 196
- “YYQRw. Format” on page 199
- “YYQRxw. Format” on page 201
- “YYQZw. Format” on page 202

YYQRw. Format

Writes SAS date values in the form [*yy*]*yy*Q*qr*, where Q is the separator, the year appears as either 2 or 4 digits, and *qr* is the quarter of the year expressed in roman numerals.

Categories: Date and Time
CAS

Alignment: Right

Syntax

YYQR*w.*

Arguments

w

specifies the width of the output field.

Default 8

Range 6–32

Interaction When the value of *w* is too small to write a four-digit year, the date appears with only the last two digits of the year.

Details

The YYQR*w*. format writes SAS date values in the form [*yy*]*yy*Q*qr*, where

<yy>yy

is a two-digit or four-digit integer that represents the year.

Q

is the character separator.

qr

is a roman numeral (I, II, III, or IV) that represents the quarter of the year.

Example

The following examples use the input value of 19537, which is the SAS date value that corresponds to June 28, 2013.

Statements	Results
`select put(19537,yyqr6.);`	`13QII`
`select put(19537,yyqr7.);`	`2013QII`
`select put(19537,yyqr.);`	`2013QII`
`select put(19537,yyqr8.);`	`2013QII`
`select put(19537,yyqr10.);`	`2013QII`

See Also

Formats:

- "YYQw. Format" on page 196

■ "YYQRxw. Format" on page 201

YYQRxw. Format

Writes date values in the form [*yy*]*yyqr* or [*yy*]*yy-qr*. The *x* in the format name is a character that represents the special character that separates the year and the quarter of the year. This character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits and *qr* is the quarter of the year in roman numerals.

Categories: Date and Time
CAS

Alignment: Right

Syntax

YYQR*xw.*

Arguments

x

identifies a separator or specifies that no separator appear between the year and the quarter. Valid values for *x* are any of the following:

C
separates with a colon

D
separates with a hyphen

N
indicates no separator

P
separates with a period

S
separates with a forward slash.

w

specifies the width of the output field.

Default 8

Range 6–32

Interactions When *x* is set to *N*, no separator is specified. The width range is then 5–32, and the default changes to 7.

When the value of *w* is too small to write a four-digit year, the date appears with only the last two digits of the year.

Details

The YYQR*xw*. format writes SAS date values in one of the following forms:

[*yy*]*yyqr*
[*yy*]*yyxqr*

where

[*yy*]*yy*
is a two-digit or four-digit integer that represents the year.

X
is a specified separator.

qr
is a roman numeral (I, II, III, or IV) that represents the quarter of the year.

Example

The following examples use the input value of 19721 which is the SAS date value that corresponds to December 29, 2013.

Statements	Results
`select put(19721,yyqrc6.);`	`13:IV`
`select put(19721,yyqrd.);`	`2013-IV`
`select put(19721,yyqrn5.);`	`13IV`
`select put(19721,yyqrp8.);`	`2013.IV`
`select put(19721,yyqrs10.);`	`2013/IV`

See Also

Formats:

- “YYQxw. Format” on page 197
- “YYQRw. Format” on page 199

YYQZw. Format

Writes SAS date values in the form [*yy*] *yyqq*. The year appears as 2 or 4 digits, and *qq* is the quarter of the year.

Categories: Date and Time
CAS

Alignment: Right

Syntax

YYQZ*w.*

Arguments

Z
specifies that no separator appear between the year and the quarter.

w
specifies the width of the output field.

Default 4

Note 6

Details

The YYQZ*w*. format writes SAS date values in the form [*yy*]*yyqq* where

[*yy*]*yy*
is a two-digit or four-digit integer that represents the year.

Z
specifies that there is no separator.

qq
is an integer (01, 02, 03, or 04) that represents the quarter of the year.

Example

The following examples use the input value of 19537, which is the SAS date value that corresponds to June 28, 2013.

Statements	Results
`select put (19537,yyqz6.);`	`201302`
`select put (19537,yyqz4.);`	`1302`

See Also

Formats:

- "DATEw. Format" on page 115
- "QTRw. Format" on page 169
- "YYQw. Format" on page 196

Zw.d Format

Writes standard numeric data with leading 0s.

Categories: Numeric
CAS

Alignment: Right

Syntax

Z*w*.[*d*]

Arguments

w

specifies the width of the output field.

Default 1

Range 1–32

Tip Allow enough space to write the value, the decimal point, and a minus sign, if necessary.

d

specifies the number of digits to the right of the decimal point in the numeric value.

Default 0

Range 0–31

Tip If *d* is 0 or you omit *d*, *Zw.d* writes the value without a decimal point.

Details

The *Zw.d* format writes standard numeric values one digit per byte and fills in 0s to the left of the data value.

The *Zw.d* format rounds to the nearest number that will fit in the output field. If *w.d* is too large to fit, SAS might shift the decimal to the BEST*w*. format. The *Zw.d* format writes negative numbers with leading minus signs. In addition, it right aligns before writing and pads the output with leading zeros.

Comparisons

The *Zw.d* format is similar to the *w.d* format except that *Zw.d* pads right-aligned output with 0s instead of blanks.

Example

The following example illustrates the *Zw.d* format:

Statement	Result
`select put(1350,z8.);`	`00001350`

See Also

Formats:

- "w.d Format" on page 179

4

FedSQL Functions

Overview of FedSQL Functions

A FedSQL function performs a computation on FedSQL expressions and returns either a single value or a set of values if the FedSQL expression is a column. Functions that perform a computation on a column are *aggregate functions*. In other SQL environments, aggregate functions are also known as set functions. Most other functions use arguments supplied by the user, but a few obtain their arguments from the operating environment.

FedSQL functions can be used in FedSQL applications or in DS2 programs where embedded FedSQL is part of a DS2 application.

For more information, see "FedSQL Expressions" on page 46 and the "SET Statement" in *SAS DS2 Language Reference*.

If the data types of the arguments in the function expression are not what is expected by the FedSQL function, FedSQL performs a type conversion on the arguments so that they have the appropriate data type. If the type conversion is successful, the function executes. Otherwise, an error is issued. For information, see "Type Conversions" on page 25.

Note: For the FedSQL functions that are like-named in Base SAS, these functions take DOUBLE and character arguments. FedSQL converts all numeric input, including ANSI standard date and time values, to DOUBLE. The date and time values become SAS date and time values. FedSQL converts all character input to CHAR. Character return values are NVARCHAR. However, the numeric return values remain as DOUBLE. In particular, date, time, and datetime values remain SAS date, time, and datetime values.

You can cast numeric output to a different data type by using the CAST function. For example, the SAS function INTNX returns a SAS datetime value as output. The

CAST function operator :: can be used to convert the output of the INTNX function to the ANSI DATE or ANSI TIMESTAMP data type as follows:

```
select intnx('month', date'2013-01-01', 5, 'beginning'):: date
select intnx('month', date'2013-01-01', 5, 'beginning'):: timestamp
```

General Function Syntax

Aggregate Function Syntax

The syntax for an aggregate function is

aggregate-function (*expression*)

aggregate-function
specifies the name of the function.

expression
specifies an expression that evaluates to a column name. The expression can include one of the following forms:

table.column
is a qualified column that identifies the table that is being processed by the FedSQL statement followed by a period and the name of the column.

Example `select avg(densities.density)from densities;`

column
is an unqualified column name that has an implicit table qualifier. The implicit table qualifier is the table that FedSQL is processing.

Example `select avg(density)from densities;`

Note: You can specify more than one argument for some aggregate functions. If you do, the Base SAS function is used and the functions can no longer be considered an aggregate function. For more information, see "Calling Base SAS Functions Instead of FedSQL Aggregate Functions" on page 219.

Non-Aggregate Function Syntax

The syntax of a function is

function (*argument* [, ...*argument*])

function
specifies the name of the function.

argument
can be a variable name, constant, or any FedSQL expression, including another function. The number and type of arguments that FedSQL allows are described with individual functions. Separate multiple arguments by a comma.

Examples

```
select sqrt(1500);

select ifnull(AvgHigh, 0) from worldtemps;
```

Using FedSQL Functions

Restrictions on Function Arguments

If the value of an argument is invalid, FedSQL sets the result to a null or missing value. Here are some common restrictions on function arguments:

- Some functions require that their arguments be restricted within a certain range. For example, the argument of the LOG function must be greater than 0.
- Most functions do not permit nulls or missing values as arguments. Exceptions include some of the descriptive statistics functions and the IFNULL function.
- In general, the allowed range of the arguments is platform-dependent, such as with the EXP function.

Using DS2 Packages in Expressions

You can invoke a DS2 package method as a function in a FedSQL SELECT statement. The syntax for the expression is shown here.

[*catalog*.] [*schema*.] *package*.*method* (*argument-1* [,… *argument-n*])

The method parameters can be a DOUBLE, DATE, TIME, TIMESTAMP, an integer of any size, or a string.

The following restrictions apply when you use DS2 packages in expressions in FedSQL:

- The method must return a value.
- You must supply a package name.
- The types and ordering of the arguments in the FedSQL SELECT statement must match the types and ordering of the parameters in the DS2 package method.

- Package methods run from PROC FEDSQL can have only input arguments in the method.

In the following example, the method **BAR** is created in DS2 and then used in a FedSQL SELECT statement call as a function.

```
/* create table */
data dataset;
   x=1; y=1; z=1; output;
   x=1; y=1; z=2; output;
   x=2; y=2; z=2; output;
   x=2; y=2; z=8; output;
   x=2; y=3; z=13; output;
run;

/*  DS2 code */
proc ds2;
   package pkga;
      method bar(double x, double y) returns double;
         return x*x + y*y;
      end;
   endpackage;
 run;
quit;

/* fedsql code */
proc fedsql;
   select pkga.bar(1,2) as five,
      cot(radians(45)) as one,
      degrees(pi()) as one_eighty,
      power(3,4) as eighty_one,
      sign(-42) as minus1,
      sign(0) as zero,
      sign(42) as plus1;
      select * from dataset where pkga.bar(x,y) = z;
quit;
```

The output from the FedSQL SELECT statement would be as follows.

FIVE	ONE	ONE_EIGHTY	EIGHTY_ONE	MINUS1	ZERO	PLUS1
5	1	180	81	-1	0	1

x	y	z
1	1	2
2	2	8
2	3	13

For more information, see "Package Method Expression" in *SAS DS2 Programmer's Guide*.

Aggregate Functions

Overview of Aggregate Functions

FedSQL aggregate functions operate on all values for an expression in a table column and return a single result. If the aggregate function is processed in a GROUP BY statement, the aggregate function returns a single result for each group. Null values and SAS missing values are not considered in the operation, except for the COUNT(*) syntax of the COUNT function. The table column that you specify in the function can be any FedSQL expression that evaluates to a column name.

The following are FedSQL aggregate functions:

Note:

Aggregate functions are currently not supported with MongoDB.

A limited set of aggregate functions are supported with Salesforce. The supported functions include AVG(), COUNT(), MIN(), MAX(), and SUM().

When you connect to Salesforce with the default SAS/ACCESS settings (which use the Salesforce SOAP API), the functions are capable of being passed down to Salesforce for processing through FedSQL pass-through. When bulk loading is

enabled with the BULK_API= option (which invokes the Salesforce Bulk API), the functions are processed in the SAS client.

Here are examples that demonstrate the functionality of the AVG, COUNT, and MAX aggregate functions on the AvgLow column of table "WorldTemps" on page 1184.

```
/* Get the average of the average low temperatures  */
select avg(AvgLow) as AvgTemp from worldtemps;

/* Get the number of different average low temperatures */
/* and group them by the average low temperature */
select AvgLow, count(AvgLow) from worldtemps group by AvgLow;

/* Get the highest average low temperature */
select max(AvgLow) from worldtemps;
```

Calling Base SAS Functions Instead of FedSQL Aggregate Functions

If multiple columns are supplied as arguments to an aggregate function and there is a like-named Base SAS function, the Base SAS function is used. The statistic that is calculated for those arguments is for the current row. The function is no longer considered to be an aggregate function. Some examples are the MIN, MAX, and SUM functions.

If multiple arguments are supplied to an aggregate function and there is no like-named Base SAS function, an error is returned. An example is the AVG function.

Function Categories

Functions can be categorized by the types of values that they operate on. Each FedSQL function belongs to one of the following categories:

Aggregate
: operates on the values in a table column.

Bitwise Logical Operations
: operates on one or more bit patterns or binary numbers at the level of their individual bits.

CAS
: identifies functions that can be used on the CAS server.

Character
: operates on character SQL expressions.

Date and Time
: operates on date and time SQL expressions.

Descriptive Statistics
: operates on values that measure central tendency, variation among values, and the shape of distribution values.

Financial
: calculates financial values such as interest, periodic payments, depreciation, and prices for European options on stocks.

Mathematical
: operates on values to perform general mathematical calculations.

Probability
: returns probability calculations.

Special
: operates on null values and SAS missing values.

Trigonometric
: operates on values to perform trigonometric calculations.

Truncation
: truncates numeric values and returns numeric values, often using fuzzing.

SAS Output Delivery System and FedSQL

The SAS Output Delivery System (ODS) rounds numeric output from functions that are submitted with PROC FEDSQL to eight characters by default. To display numeric output with the full precision of which FedSQL is capable, use the PUT function with the BEST16. format, as follows:

```
select PUT (beta(5,3), best16.) as Beta;
```

FEDSQL Functions by Category

Category	Language Elements	Description
Aggregate	AVG Function (p. 290)	Returns the average of all values in a column.
	COUNT Function (p. 391)	Returns the number of rows retrieved by a SELECT statement for a specified table.
	CSS Function (p. 401)	Returns the corrected sum of squares of all values in an expression.
	KURTOSIS Function (p. 609)	Returns the kurtosis of all values in an expression.

Category	Language Elements	Description
	MAX Function (p. 642)	Returns the maximum value in a column.
	MIN Function (p. 648)	Returns the minimum value in an expression.
	NMISS Function (p. 661)	Returns the number of null values or SAS missing values in an expression.
	PROBT Function (p. 768)	Returns the probability from a t distribution of the values in an expression.
	RANGE Function (p. 780)	Returns the range between values in an expression.
	SKEWNESS Function (p. 811)	Returns the skewness of all values in an expression.
	STDDEV Function (p. 819)	Returns the statistical standard deviation of all values in an expression.
	STDERR Function (p. 820)	Returns the statistical standard error of all values in an expression.
	STUDENTS_T Function (p. 822)	Returns the Student's t distribution of the values in an expression.
	SUM Function (p. 825)	Returns the sum of all the values in an expression.
	USS Function (p. 851)	Returns the uncorrected sum of squares of all the values in an expression.
	VARIANCE Function (p. 854)	Returns the measure of the dispersion of all values in an expression.
Bitwise Logical Operations	BAND Function (p. 291)	Returns the bitwise logical AND of two arguments.
	BLSHIFT Function (p. 304)	Returns the bitwise logical left shift of two arguments.
	BNOT Function (p. 305)	Returns the bitwise logical NOT of an argument.
	BOR Function (p. 306)	Returns the bitwise logical OR of two arguments.
	BRSHIFT Function (p. 307)	Returns the bitwise logical right shift of two arguments.
	BXOR Function (p. 308)	Returns the bitwise logical EXCLUSIVE OR of two arguments.
CAS	ABS Function (p. 259)	Returns the absolute value of a numeric value expression.
	AIRY Function (p. 260)	Returns the value of the Airy function.
	ANYALNUM Function (p. 261)	Searches a character string for an alphanumeric character, and returns the first character position at which the character is found.

Category	Language Elements	Description
	ANYALPHA Function (p. 263)	Searches a character string for an alphabetic character, and returns the first character position at which the character is found.
	ANYDIGIT Function (p. 265)	Searches a character string for a digit, and returns the first character position at which the digit is found.
	ANYFIRST Function (p. 267)	Searches a character string for a character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.
	ANYLOWER Function (p. 269)	Searches a character string for a lowercase letter, and returns the first character position at which the letter is found.
	ANYNAME Function (p. 271)	Searches a character string for a character that is valid in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.
	ANYPUNCT Function (p. 273)	Searches a character string for a punctuation character, and returns the first character position at which that character is found.
	ANYSPACE Function (p. 275)	Searches a character string for a whitespace character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first character position at which that character is found.
	ANYUPPER Function (p. 276)	Searches a character string for an uppercase letter, and returns the first character position at which the letter is found.
	ANYXDIGIT Function (p. 278)	Searches a character string for a hexadecimal character that represents a digit, and returns the first character position at which that character is found.
	ARCOS Function (p. 280)	Returns the arccosine in radians.
	ARCOSH Function (p. 281)	Returns the inverse hyperbolic cosine.
	ARSIN Function (p. 283)	Returns the arcsine in radians.
	ARSINH Function (p. 284)	Returns the inverse hyperbolic sine.
	ARTANH Function (p. 285)	Returns the inverse hyperbolic tangent.
	ATAN Function (p. 287)	Returns the arctangent in radians.
	ATAN2 Function (p. 288)	Returns the arctangent of the x and y coordinates of a right triangle, in radians.
	AVG Function (p. 290)	Returns the average of all values in a column.

Category	Language Elements	Description
	BAND Function (p. 291)	Returns the bitwise logical AND of two arguments.
	BETA Function (p. 292)	Returns the value of the beta function.
	BETAINV Function (p. 293)	Returns a quantile from the beta distribution.
	BLACKCLPRC Function (p. 295)	Calculates call prices for European options on futures, based on the Black model.
	BLACKPTPRC Function (p. 297)	Calculates put prices for European options on futures, based on the Black model.
	BLKSHCLPRC Function (p. 299)	Calculates call prices for European options on stocks, based on the Black-Scholes model.
	BLKSHPTPRC Function (p. 302)	Calculates put prices for European options on stocks, based on the Black-Scholes model.
	BLSHIFT Function (p. 304)	Returns the bitwise logical left shift of two arguments.
	BNOT Function (p. 305)	Returns the bitwise logical NOT of an argument.
	BOR Function (p. 306)	Returns the bitwise logical OR of two arguments.
	BRSHIFT Function (p. 307)	Returns the bitwise logical right shift of two arguments.
	BXOR Function (p. 308)	Returns the bitwise logical EXCLUSIVE OR of two arguments.
	BYTE Function (p. 309)	Returns one character in the ASCII or the EBCDIC collating sequence.
	CDF Function (p. 312)	Computes the left cumulative distribution function from various continuous and discrete probability distributions.
	CDF BERNOULLI Distribution Function (p. 315)	Returns a value from the Bernoulli cumulative probability distribution.
	CDF BETA Distribution Function (p. 316)	Returns a value from the beta cumulative probability distribution.
	CDF BINOMIAL Distribution Function (p. 318)	Returns a value from the binomial cumulative probability distribution.
	CDF CAUCHY Distribution Function (p. 320)	Returns a value from the Cauchy cumulative probability distribution.
	CDF Chi-Square Distribution Function (p. 322)	Returns a value from the chi-square cumulative probability distribution.
	CDF Conway-Maxwell-Poisson Distribution Function (p. 324)	Returns a value from the Conway-Maxwell-Poisson cumulative probability distribution.

Category	Language Elements	Description
	CDF Exponential Distribution Function (p. 325)	Returns a value from the exponential cumulative probability distribution.
	CDF F Distribution Function (p. 327)	Returns a value from the F cumulative probability distribution.
	CDF GAMMA Distribution Function (p. 329)	Returns a value from the gamma cumulative probability distribution.
	CDF Generalized Poisson Distribution Function (p. 330)	Returns a value from the generalized Poisson cumulative probability distribution.
	CDF GEOMETRIC Distribution Function (p. 332)	Returns a value from the geometric cumulative probability distribution.
	CDF HYPERGEOMETRIC Distribution Function (p. 333)	Returns a value from the hypergeometric cumulative probability distribution.
	CDF LAPLACE Distribution Function (p. 335)	Returns a value from the Laplace cumulative probability distribution.
	CDF LOGISTIC Distribution Function (p. 337)	Returns a value from the logistic cumulative probability distribution.
	CDF LOGNORMAL Distribution Function (p. 339)	Returns a value from the lognormal cumulative probability distribution.
	CDF NEGBINOMIAL Distribution Function (p. 340)	Returns a value from the negative binomial cumulative probability distribution.
	CDF NORMAL Distribution Function (p. 342)	Returns a value from the normal cumulative probability distribution.
	CDF NORMALMIX Distribution Function (p. 344)	Returns a value from the normal mixture cumulative probability distribution.
	CDF PARETO Distribution Function (p. 346)	Returns a value from the Pareto cumulative probability distribution.
	CDF POISSON Distribution Function (p. 347)	Returns a value from the Poisson cumulative probability distribution.
	CDF T Distribution Function (p. 349)	Returns a value from the T cumulative probability distribution.
	CDF TWEEDIE Distribution Function (p. 351)	Returns a value from the Tweedie cumulative probability distribution.
	CDF UNIFORM Distribution Function (p. 353)	Returns a value from the uniform cumulative probability distribution.
	CDF WALD (Inverse Gaussian) Distribution Function (p. 354)	Returns a value from the Wald (also known as the inverse Gaussian) cumulative probability distribution.

Category	Language Elements	Description
	CDF WEIBULL Distribution Function (p. 356)	Returns a value from the Weibull cumulative probability distribution.
	CEIL Function (p. 358)	Returns the smallest integer greater than or equal to a numeric value expression.
	CEILZ Function (p. 359)	Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing.
	CHARACTER_LENGTH Function (p. 361)	Returns the number of characters in a string of any data type.
	CNONCT Function (p. 362)	Returns the noncentrality parameter from a chi-square distribution.
	COALESCE Function (p. 364)	Returns the first non-null or nonmissing value from a list of numeric arguments.
	COALESCEC Function (p. 366)	Returns the first non-null or nonmissing value from a list of character arguments.
	COMB Function (p. 367)	Computes the number of combinations of n elements taken r at a time.
	COMPARE Function (p. 368)	Returns the position of the leftmost character by which two strings differ, or returns 0 if there is no difference.
	COMPBL Function (p. 372)	Removes multiple blanks from a character string.
	COMPOUND Function (p. 374)	Returns compound interest parameters.
	COMPRESS Function (p. 375)	Returns a character string with specified characters removed from the original string.
	CONSTANT Function (p. 378)	Computes machine and mathematical constants.
	CONVX Function (p. 384)	Returns the convexity for an enumerated cash flow.
	CONVXP Function (p. 386)	Returns the convexity for a periodic cash flow stream, such as a bond.
	COS Function (p. 388)	Returns the cosine in radians.
	COSH Function (p. 389)	Returns the hyperbolic cosine in radians.
	COT Function (p. 390)	Returns the tangent in radians.
	COUNT Function (p. 391)	Returns the number of rows retrieved by a SELECT statement for a specified table.
	COUNTC Function (p. 393)	Counts the number of characters in a string that appear or do not appear in a list of characters.

Category	Language Elements	Description
	COUNTW Function (p. 396)	Counts the number of words in a character string.
	CSC Function (p. 400)	Returns the cosecant.
	CSS Function (p. 401)	Returns the corrected sum of squares of all values in an expression.
	CUMIPMT Function (p. 403)	Returns the cumulative interest paid on a loan between the start and end period.
	CUMPRINC Function (p. 404)	Returns the cumulative principal paid on a loan between the start and end period.
	CURRENT_DATE Function (p. 406)	Returns the current date for the time zone.
	CURRENT_LOCALE Function (p. 407)	Returns the five character name of the current locale.
	CURRENT_TIME Function (p. 407)	Returns the current time for your time zone.
	CURRENT_TIME_GMT Function (p. 408)	Returns the current GMT time.
	CURRENT_TIMESTAMP Function (p. 409)	Returns the date and time for your time zone.
	CURRENT_TIMESTAMP_GMT Function (p. 410)	Returns the current GMT date and time.
	CV Function (p. 411)	Returns the coefficient of variation.
	DAIRY Function (p. 412)	Returns the derivative of the AIRY function.
	DATDIF Function (p. 413)	Returns the number of days between two dates after computing the difference between the dates according to specified day count conventions.
	DATE Function (p. 416)	Returns the current date as a SAS date value.
	DATEJUL Function (p. 417)	Converts a Julian date to a SAS date value.
	DATEPART Function (p. 418)	Returns the date as year, month, and day.
	DATETIME Function (p. 419)	Returns the current date and time of day as a SAS datetime value.
	DAY Function (p. 420)	Returns the numeric day of the month from a date or datetime value.
	DEGREES Function (p. 421)	Returns the number of degrees for an angle in radians.

Category	Language Elements	Description
	DEQUOTE Function (p. 423)	Removes matching single quotation marks from a character string that begins with a single quotation mark, and deletes all characters to the right of the closing quotation mark.
	DEVIANCE Function (p. 425)	Returns the deviance based on a probability distribution.
	DHMS Function (p. 429)	Returns a SAS datetime value from date, hour, minute, and second values.
	DIGAMMA Function (p. 431)	Returns the value of the digamma function.
	DUR Function (p. 432)	Returns the modified duration for an enumerated cash flow.
	DURP Function (p. 434)	Returns the modified duration for a periodic cash flow stream, such as a bond.
	E Function (p. 436)	Returns the natural logarithm, e.
	EFFRATE Function (p. 437)	Returns the effective annual interest rate.
	ERF Function (p. 438)	Returns the value of the (normal) error function.
	ERFC Function (p. 439)	Returns the value of the complementary (normal) error function.
	EXP Function (p. 441)	Returns the value of the e constant raised to a specified power.
	FACT Function (p. 442)	Computes a factorial.
	FINANCE Function (p. 443)	Computes financial calculations such as depreciation, maturation, accrued interest, net present value, periodic savings, and internal rates of return.
	FINANCE ACCRINT Function (p. 447)	Computes the accrued interest for a security that pays periodic interest.
	FINANCE ACCRINTM Function (p. 449)	Computes the accrued interest for a security that pays interest at maturity.
	FINANCE AMORDEGRC Function (p. 451)	Computes the depreciation for each accounting period by using a depreciation coefficient.
	FINANCE AMORLINC Function (p. 452)	Computes the depreciation for each accounting period.
	FINANCE COUPDAYBS Function (p. 454)	Computes the number of days from the beginning of the coupon period to the settlement date.
	FINANCE COUPDAYS Function (p. 456)	Computes the number of days in the coupon period that contains the settlement date.
	FINANCE COUPDAYSNC Function (p. 457)	Computes the number of days from the settlement date to the next coupon date.

Category	Language Elements	Description
	FINANCE COUPNCD Function (p. 459)	Computes the next coupon date after the settlement date.
	FINANCE COUPNUM Function (p. 460)	Computes the number of coupons that are payable between the settlement date and the maturity date.
	FINANCE COUPPCD Function (p. 462)	Computes the previous coupon date before the settlement date.
	FINANCE CUMIPMT Function (p. 463)	Computes the cumulative interest paid between two periods.
	FINANCE CUMPRINC Function (p. 465)	Computes the cumulative principal that is paid on a loan between two periods.
	FINANCE DB Function (p. 466)	Computes the depreciation of an asset for a specified period by using the fixed-declining balance method.
	FINANCE DDB Function (p. 467)	Computes the depreciation of an asset for a specified period by using the double-declining balance method or some other method that you specify.
	FINANCE DISC Function (p. 468)	Computes the discount rate for a security.
	FINANCE DOLLARDE Function (p. 470)	Converts a dollar price, expressed as a fraction, to a dollar price, expressed as a decimal number.
	FINANCE DOLLARFR Function (p. 471)	Converts a dollar price, expressed as a decimal number, to a dollar price, expressed as a fraction.
	FINANCE DURATION Function (p. 472)	Computes the annual duration of a security with periodic interest payments.
	FINANCE EFFECT Function (p. 473)	Computes the effective annual interest rate.
	FINANCE FV Function (p. 474)	Computes the future value of an investment.
	FINANCE FVSCHEDULE Function (p. 475)	Computes the future value of the initial principal after applying a series of compound interest rates.
	FINANCE INTRATE Function (p. 476)	Computes the interest rate for a fully invested security.
	FINANCE IPMT Function (p. 478)	Computes the interest payment for an investment for a specified period.
	FINANCE IRR Function (p. 479)	Computes the internal rate of return for a series of cash flows.
	FINANCE ISPMT Function (p. 480)	Calculates the interest paid during a specific period of an investment.

Category	Language Elements	Description
	FINANCE MDURATION Function (p. 481)	Computes the Macaulay modified duration for a security with an assumed face value of $100.
	FINANCE MIRR Function (p. 483)	Computes the internal rate of return where positive and negative cash flows are financed at different rates.
	FINANCE NOMINAL Function (p. 484)	Computes the annual nominal interest rates.
	FINANCE NPER Function (p. 485)	Computes the number of periods for an investment.
	FINANCE NPV Function (p. 486)	Computes the net present value of an investment based on a series of periodic cash flows and a discount rate.
	FINANCE ODDFPRICE Function (p. 487)	Computes the price of a security per $100 face value with an odd first period.
	FINANCE ODDFYIELD Function (p. 489)	Computes the yield of a security with an odd first period.
	FINANCE ODDLPRICE Function (p. 491)	Computes the price of a security per $100 face value with an odd last period.
	FINANCE ODDLYIELD Function (p. 493)	Computes the yield of a security with an odd last period.
	FINANCE PMT Function (p. 495)	Computes the periodic payment of an annuity.
	FINANCE PPMT Function (p. 496)	Computes the payment on the principal for an investment for a specified period.
	FINANCE PRICE Function (p. 497)	Computes the price of a security per $100 face value that pays periodic interest.
	FINANCE PRICEDISC Function (p. 499)	Computes the price of a discounted security per $100 face value.
	FINANCE PRICEMAT Function (p. 500)	Computes the price of a security per $100 face value that pays interest at maturity.
	FINANCE PV Function (p. 502)	Computes the present value of an investment.
	FINANCE RATE Function (p. 503)	Computes the interest rate per period of an annuity.
	FINANCE RECEIVED Function (p. 505)	Computes the amount that is received at maturity for a fully invested security.
	FINANCE SLN Function (p. 506)	Computes the straight-line depreciation of an asset for one period.

Category	Language Elements	Description
	FINANCE SYD Function (p. 507)	Computes the sum-of-years digits depreciation of an asset for a specified period.
	FINANCE TBILLEQ Function (p. 508)	Computes the bond-equivalent yield for a treasury bill.
	FINANCE TBILLPRICE Function (p. 509)	Computes the price of a treasury bill per $100 face value.
	FINANCE TBILLYIELD Function (p. 510)	Computes the yield for a treasury bill.
	FINANCE VDB Function (p. 511)	Computes the depreciation of an asset for a specified or partial period by using a declining balance method.
	FINANCE XIRR Function (p. 513)	Computes the internal rate of return for a schedule of cash flows that is not necessarily periodic.
	FINANCE XNPV Function (p. 514)	Computes the net present value for a schedule of cash flows that is not necessarily periodic.
	FINANCE YIELD Function (p. 515)	Computes the yield on a security that pays periodic interest.
	FINANCE YIELDDISC Function (p. 517)	Computes the annual yield for a discounted security (for example, a treasury bill).
	FINANCE YIELDMAT Function (p. 518)	Computes the annual yield of a security that pays interest at maturity.
	FLOOR Function (p. 520)	Returns the largest integer less than or equal to a numeric value expression.
	FLOORZ Function (p. 521)	Returns the largest integer that is less than or equal to the argument, using zero fuzzing.
	FMTINFO Function (p. 523)	Returns information about a SAS format or informat.
	FNONCT Function (p. 525)	Returns the value of the noncentrality parameter of an F distribution.
	FUZZ Function (p. 527)	Returns the nearest whole number if the argument is within 1E-12 of that number.
	GAMINV Function (p. 528)	Returns a quantile from the gamma distribution.
	GAMMA Function (p. 530)	Returns the value of the gamma function.
	GARKHCLPRC Function (p. 531)	Calculates call prices for European options on stocks, based on the Garman-Kohlhagen model.
	GARKHPTPRC Function (p. 533)	Calculates put prices for European options on stocks, based on the Garman-Kohlhagen model.

Category	Language Elements	Description
	GCD Function (p. 536)	Returns the greatest common divisor for a set of integers.
	GEOMEAN Function (p. 537)	Returns the geometric mean.
	GEOMEANZ Function (p. 539)	Returns the geometric mean, using zero fuzzing.
	HARMEAN Function (p. 540)	Returns the harmonic mean.
	HARMEANZ Function (p. 542)	Returns the harmonic mean, using zero fuzzing.
	HMS Function (p. 544)	Returns a SAS time value from hour, minute, and second values.
	HOLIDAY Function (p. 545)	Returns a SAS date value of a specified holiday for a specified year.
	HOUR Function (p. 549)	Returns the hour from a time or datetime value.
	IBESSEL Function (p. 550)	Returns the value of the modified Bessel function.
	IFNULL Function (p. 551)	Checks the value of the first expression and, if it is null or a SAS missing value, returns the second expression.
	INDEX Function (p. 553)	Searches a character expression for a string of characters, and returns the position of the string's first character for the first occurrence of the string.
	INDEXC Function (p. 555)	Searches a character expression for specified characters and returns the position of the first occurrence of any of the characters.
	INPUTC Function (p. 556)	Enables you to specify a character informat at run time.
	INPUTN Function (p. 558)	Enables you to specify a numeric informat at run time.
	INT Function (p. 559)	Returns the whole number, fuzzed to avoid unexpected floating-point results.
	INTCINDEX Function (p. 561)	Returns the cycle index when a date, time, or timestamp interval and value are specified.
	INTCK Function (p. 564)	Returns the number of interval boundaries of a given kind that lie between two SAS dates, times, or timestamp values encoded as DOUBLE.
	INTCYCLE Function (p. 571)	Returns the date, time, or datetime interval at the next higher seasonal cycle when a date, time, or datetime interval is specified.
	INTFIT Function (p. 574)	Returns a time interval that is aligned between two dates.

Category	Language Elements	Description
	INTGET Function (p. 575)	Returns a time interval based on three date or datetime values.
	INTINDEX Function (p. 578)	Returns the seasonal index when a date, time, or timestamp interval and value are specified.
	INTNX Function (p. 582)	Increments a SAS date, time, or datetime value encoded as a DOUBLE, and returns a SAS date, time, or datetime value encoded as a DOUBLE.
	INTRR Function (p. 588)	Returns the internal rate of return as a decimal value.
	INTSEAS Function (p. 590)	Returns the length of the seasonal cycle when a date, time, or datetime interval is specified.
	INTSHIFT Function (p. 594)	Returns the shift interval that corresponds to the base interval.
	INTTEST Function (p. 596)	Returns 1 if a time interval is valid, and returns 0 if a time interval is invalid.
	INTZ Function (p. 598)	Returns the whole number portion of the argument, using zero fuzzing.
	IPMT Function (p. 600)	Returns the interest payment for a given period for a constant payment loan or the periodic savings for a future balance.
	IQR Function (p. 602)	Returns the interquartile range.
	IRR Function (p. 603)	Returns the internal rate of return as a percentage.
	JBESSEL Function (p. 605)	Returns the value of the Bessel function.
	JULDATE Function (p. 606)	Returns the Julian date from a SAS date value.
	JULDATE7 Function (p. 607)	Returns a seven-digit Julian date from a SAS date value.
	KURTOSIS Function (p. 609)	Returns the kurtosis of all values in an expression.
	LARGEST Function (p. 610)	Returns the kth largest non-null or nonmissing value.
	LCM Function (p. 612)	Returns the least common multiple for a set of whole numbers.
	LCOMB Function (p. 613)	Computes the logarithm of the COMB function, which is the logarithm of the number of combinations of n objects taken r at a time.
	LEFT Function (p. 614)	Left aligns a character expression.
	LFACT Function (p. 615)	Computes the logarithm of the FACT (factorial) function.
	LGAMMA Function (p. 616)	Returns the natural logarithm of the Gamma function.

Category	Language Elements	Description
	LOG Function (p. 617)	Returns the natural logarithm (base e) of a numeric value expression.
	LOG1PX Function (p. 618)	Returns the log of 1 plus the argument.
	LOG2 Function (p. 619)	Returns the base-2 logarithm of a numeric value expression.
	LOG10 Function (p. 620)	Returns the base-10 logarithm of a numeric value expression.
	LOGBETA Function (p. 621)	Returns the logarithm of the beta function.
	LOGCDF Function (p. 622)	Returns the logarithm of a left cumulative distribution function.
	LOGISTIC Function (p. 625)	Returns the logistic transformation of the argument.
	LOGPDF Function (p. 626)	Computes the logarithm of the probability density (mass) function from various continuous and discrete distributions.
	LOGSDF Function (p. 628)	Returns the logarithm of a survival function.
	LOWCASE Function (p. 631)	Converts all letters in a character expression to lowercase.
	MAD Function (p. 632)	Returns the median absolute deviation from the median.
	MAKEDATE Function (p. 633)	Returns the date as year, month, and day.
	MAKETIME Function (p. 634)	Returns the time as hours, minutes, and seconds.
	MAKETIMESTAMP Function (p. 635)	Returns the timestamp.
	MARGRCLPRC Function (p. 636)	Calculates call prices for European options on stocks, based on the Margrabe model.
	MARGRPTPRC Function (p. 639)	Calculates put prices for European options on stocks, based on the Margrabe model.
	MAX Function (p. 642)	Returns the maximum value in a column.
	MDY Function (p. 644)	Returns a SAS date value from month, day, and year values.
	MEAN Function (p. 645)	Returns the arithmetic mean (average) of the non-null or nonmissing arguments.
	MEDIAN Function (p. 646)	Returns the median value.
	MIN Function (p. 648)	Returns the minimum value in an expression.
	MINUTE Function (p. 649)	Returns the minute from a time or datetime value.

Category	Language Elements	Description
	MOD Function (p. 651)	Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid most unexpected floating-point results.
	MODZ Function (p. 653)	Returns the remainder from the division of the first argument by the second argument, using zero fuzzing.
	MONTH Function (p. 655)	Returns the numeric month from a date or datetime value.
	MORT Function (p. 656)	Returns amortization parameters.
	N Function (p. 658)	Returns the number of non-null or nonmissing numeric values.
	NETPV Function (p. 660)	Returns the net present value as a percent.
	NMISS Function (p. 661)	Returns the number of null values or SAS missing values in an expression.
	NOMRATE Function (p. 664)	Returns the nominal annual interest rate.
	NOTALNUM Function (p. 665)	Searches a character string for a non-alphanumeric character, and returns the first character position at which the character is found.
	NOTALPHA Function (p. 667)	Searches a character string for a nonalphabetic character, and returns the first character position at which the character is found.
	NOTDIGIT Function (p. 669)	Searches a character string for any character that is not a digit, and returns the first character position at which that character is found.
	NOTFIRST Function (p. 671)	Searches a character string for an invalid first character in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.
	NOTLOWER Function (p. 673)	Searches a character string for a character that is not a lowercase letter, and returns the first character position at which that character is found.
	NOTNAME Function (p. 675)	Searches a character string for an invalid character in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.
	NOTPUNCT Function (p. 677)	Searches a character string for a character that is not a punctuation character, and returns the first character position at which that character is found.
	NOTSPACE Function (p. 679)	Searches a character string for a character that is not a whitespace character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first character position at which that character is found.

Category	Language Elements	Description
	NOTUPPER Function (p. 681)	Searches a character string for a character that is not an uppercase letter, and returns the first character position at which that character is found.
	NOTXDIGIT Function (p. 683)	Searches a character string for a character that is not a hexadecimal character, and returns the first character position at which that character is found.
	NPV Function (p. 685)	Returns the net present value with the rate expressed as a percentage.
	NWKDOM Function (p. 686)	Returns the date for the nth occurrence of a weekday for the specified month and year.
	OCTET_LENGTH Function (p. 689)	Returns the number of bytes in a string of any data type.
	ORDINAL Function (p. 690)	Orders a list of values, and returns a value that is based on a position in the list.
	PCTL Function (p. 691)	Returns the percentile that corresponds to the percentage.
	PDF Function (p. 693)	Returns a value from a probability density (mass) distribution.
	PDF BERNOULLI Distribution Function (p. 695)	Returns a value from the Bernoulli probability density (mass) distribution.
	PDF BETA Distribution Function (p. 697)	Returns a value from the beta probability density (mass) distribution.
	PDF BINOMIAL Distribution Function (p. 699)	Returns a value from the binomial probability density (mass) distribution.
	PDF CAUCHY Distribution Function (p. 700)	Returns a value from the Cauchy probability density (mass) distribution.
	PDF Chi-Square Distribution Function (p. 702)	Returns a value from the chi-square probability density (mass) distribution.
	PDF Conway-Maxwell-Poisson Distribution Function (p. 704)	Returns a value from the Conway-Maxwell-Poisson probability density (mass) distribution.
	PDF EXPONENTIAL Distribution Function (p. 706)	Returns a value from the exponential probability density (mass) distribution.
	PDF F Distribution Function (p. 708)	Returns a value from the F probability density (mass) distribution.
	PDF GAMMA Distribution Function (p. 710)	Returns a value from the gamma probability density (mass) distribution.

Category	Language Elements	Description
	PDF Generalized Poisson Distribution Function (p. 711)	Returns a value from the generalized Poisson probability density (mass) distribution.
	PDF GEOMETRIC Distribution Function (p. 713)	Returns a value from the geometric probability density (mass) distribution.
	PDF Hypergeometric Distribution Function (p. 714)	Returns a value from a hypergeometric probability density (mass) distribution.
	PDF LAPLACE Distribution Function (p. 716)	Returns a value from the Laplace probability density (mass) distribution.
	PDF LOGISTIC Distribution Function (p. 718)	Returns a value from the logistic probability density (mass) distribution.
	PDF LOGNORMAL Distribution Function (p. 720)	Returns a value from the lognormal probability density (mass) distribution.
	PDF NEGBINOMIAL Distribution Function (p. 721)	Returns the value from the negative binomial probability density (mass) distribution.
	PDF NORMAL Distribution Function (p. 723)	Returns a value from the normal probability density (mass) distribution.
	PDF NORMALMIX Distribution Function (p. 724)	Returns a value from the normal mixture probability density (mass) distribution.
	PDF PARETO Distribution Function (p. 726)	Returns a value from the Pareto probability density (mass) distribution.
	PDF POISSON Distribution Function (p. 728)	Returns a value from the Poisson probability density (mass) distribution.
	PDF T Distribution Function (p. 729)	Returns a value from the T probability density (mass) distribution.
	PDF TWEEDIE Distribution Function (p. 731)	Returns a value from the Tweedie probability density (mass) distribution.
	PDF UNIFORM Distribution Function (p. 733)	Returns a value from the uniform probability density (mass) distribution.
	PDF Wald (Inverse Gaussian) Distribution Function (p. 735)	Returns a value from the Wald (also known as the inverse Gaussian) probability density (mass) distribution.
	PDF WEIBULL Distribution Function (p. 736)	Returns a value from the Weibull probability density (mass) distribution.
	PERM Function (p. 738)	Computes the number of permutations of n items that are taken r at a time.
	PI Function (p. 739)	Returns the constant value of PI as a floating-point value.

Category	Language Elements	Description
	PMT Function (p. 740)	Returns the periodic payment for a constant payment loan or the periodic savings for a future balance.
	POISSON Function (p. 742)	Returns the probability from a Poisson distribution.
	POWER Function (p. 743)	Returns the value of a numeric value expression raised to a specified power.
	PROBBETA Function (p. 744)	Returns the probability from a beta distribution.
	PROBBNML Function (p. 745)	Returns the probability from a binomial distribution.
	PROBBNRM Function (p. 746)	Returns a probability from a bivariate normal distribution.
	PROBCHI Function (p. 748)	Returns the probability from a chi-square distribution.
	PROBF Function (p. 749)	Returns the probability from an F distribution.
	PROBGAM Function (p. 750)	Returns the probability from a gamma distribution.
	PROBHYPR Function (p. 751)	Returns the probability from a hypergeometric distribution.
	PROBIT Function (p. 753)	Returns a quantile from the standard normal distribution.
	PROBMC Function (p. 754)	Returns a probability or a quantile from various distributions for multiple comparisons of means.
	PROBMED Function (p. 764)	Computes cumulative probabilities for the sample median.
	PROBNEGB Function (p. 765)	Returns the probability from a negative binomial distribution.
	PROBNORM Function (p. 767)	Returns the probability from the standard normal distribution.
	PROBT Function (p. 768)	Returns the probability from a t distribution of the values in an expression.
	PUT Function (p. 769)	Returns a value using a specified format.
	PVP Function (p. 771)	Returns the present value for a periodic cash flow stream (such as a bond), with repayment of principal at maturity.
	QTR Function (p. 773)	Returns the quarter of the year from a SAS date value.
	QUANTILE Function (p. 774)	Returns the quantile from a distribution when you specify the left probability (CDF).
	QUOTE Function (p. 778)	Adds double quotation marks to a character value.

Category	Language Elements	Description
	RADIANS Function (p. 779)	Returns the number of radians converted from a numeric degree value.
	RANGE Function (p. 780)	Returns the range between values in an expression.
	RANK Function (p. 782)	Returns the position of a character in the ASCII collating sequence.
	REPEAT Function (p. 783)	Repeats a character expression.
	REVERSE Function (p. 784)	Reverses a character expression.
	RMS Function (p. 785)	Returns the root mean square.
	ROUND Function (p. 786)	Rounds the first argument to the nearest multiple of the second argument, or to the nearest integer when the second argument is omitted.
	ROUNDZ Function (p. 789)	Rounds the first argument to the nearest multiple of the second argument, using zero fuzzing.
	SAVINGS Function (p. 792)	Returns the balance of a periodic savings by using variable interest rates.
	SCAN Function (p. 795)	Returns the nth word from a character expression.
	SDF Function (p. 801)	Returns a survival function.
	SEC Function (p. 804)	Returns the secant.
	SECOND Function (p. 806)	Returns the second from a time or datetime value.
	SIGN Function (p. 807)	Returns a number that indicates the sign of a numeric value expression.
	SIN Function (p. 808)	Returns the trigonometric sine.
	SINH Function (p. 809)	Returns the hyperbolic sine.
	SKEWNESS Function (p. 811)	Returns the skewness of all values in an expression.
	SMALLEST Function (p. 812)	Returns the kth smallest non-null or nonmissing value.
	SQUANTILE Function (p. 814)	Returns the quantile from a distribution when you specify the right probability (SDF).
	SQRT Function (p. 817)	Returns the square root of a value.
	STD Function (p. 818)	Returns the standard deviation.
	STDDEV Function (p. 819)	Returns the statistical standard deviation of all values in an expression.

Category	Language Elements	Description
	STDERR Function (p. 820)	Returns the statistical standard error of all values in an expression.
	STUDENTS_T Function (p. 822)	Returns the Student's t distribution of the values in an expression.
	SUBSTRING Function (p. 823)	Extracts a substring from a character string.
	SUM Function (p. 825)	Returns the sum of all the values in an expression.
	SUMABS Function (p. 826)	Returns the sum of the absolute values of the nonmissing arguments.
	SYSGET Function (p. 827)	Returns the value of the specified operating environment variable.
	TAN Function (p. 829)	Returns the tangent.
	TANH Function (p. 830)	Returns the hyperbolic tangent.
	TIMEPART Function (p. 831)	Returns the time as hours, minutes, and seconds.
	TIMEVALUE Function (p. 832)	Returns the equivalent of a reference amount at a base date by using variable interest rates.
	TINV Function (p. 834)	Returns a quantile from the t distribution.
	TNONCT Function (p. 836)	Returns the value of the noncentrality parameter from the Student's t distribution.
	TODAY Function (p. 838)	Returns the current date as a numeric SAS date value.
	TRANSTRN Function (p. 839)	Replaces or removes all occurrences of a substring in a character string.
	TRANWRD Function (p. 841)	Replaces or removes all occurrences of a substring in a character string.
	TRIGAMMA Function (p. 843)	Returns the value of the trigamma function.
	TRIM Function (p. 844)	Removes leading characters, trailing characters, or both from a character string.
	TRUNC Function (p. 846)	Truncates a numeric value to a specified length.
	UPCASE Function (p. 847)	Converts all letters in an argument to uppercase.
	URLDECODE Function (p. 848)	Returns a string that was decoded using the URL escape syntax.
	URLENCODE Function (p. 850)	Returns a string that was encoded using the URL escape syntax.

Category	Language Elements	Description
	USS Function (p. 851)	Returns the uncorrected sum of squares of all the values in an expression.
	VAR Function (p. 853)	Returns the variance.
	VARIANCE Function (p. 854)	Returns the measure of the dispersion of all values in an expression.
	VERIFY Function (p. 855)	Returns the position of the first character that is unique to an expression.
	WEEK Function (p. 857)	Returns the week-number value.
	WEEKDAY Function (p. 860)	From a SAS date value, returns a whole number that corresponds to the day of the week.
	WHICHC Function (p. 861)	Returns the first position of a character string from a list of character strings.
	WHICHN Function (p. 863)	Returns the first position of a number from a list of numbers.
	YEAR Function (p. 864)	Returns the year from a date or datetime value.
	YIELDP Function (p. 866)	Returns the yield-to-maturity for a periodic cash flow stream, such as a bond.
	YRDIF Function (p. 867)	Returns the difference in years between two dates according to specified day count conventions; returns a person's age.
	YYQ Function (p. 870)	Returns a SAS date value from year and quarter year values.
Character	ANYALNUM Function (p. 261)	Searches a character string for an alphanumeric character, and returns the first character position at which the character is found.
	ANYALPHA Function (p. 263)	Searches a character string for an alphabetic character, and returns the first character position at which the character is found.
	ANYDIGIT Function (p. 265)	Searches a character string for a digit, and returns the first character position at which the digit is found.
	ANYFIRST Function (p. 267)	Searches a character string for a character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.
	ANYLOWER Function (p. 269)	Searches a character string for a lowercase letter, and returns the first character position at which the letter is found.
	ANYNAME Function (p. 271)	Searches a character string for a character that is valid in a SAS variable name under VALIDVARNAME=V7, and

Category	Language Elements	Description
		returns the first character position at which that character is found.
	ANYPUNCT Function (p. 273)	Searches a character string for a punctuation character, and returns the first character position at which that character is found.
	ANYSPACE Function (p. 275)	Searches a character string for a whitespace character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first character position at which that character is found.
	ANYUPPER Function (p. 276)	Searches a character string for an uppercase letter, and returns the first character position at which the letter is found.
	ANYXDIGIT Function (p. 278)	Searches a character string for a hexadecimal character that represents a digit, and returns the first character position at which that character is found.
	BYTE Function (p. 309)	Returns one character in the ASCII or the EBCDIC collating sequence.
	CHARACTER_LENGTH Function (p. 361)	Returns the number of characters in a string of any data type.
	COALESCEC Function (p. 366)	Returns the first non-null or nonmissing value from a list of character arguments.
	COMPARE Function (p. 368)	Returns the position of the leftmost character by which two strings differ, or returns 0 if there is no difference.
	COMPBL Function (p. 372)	Removes multiple blanks from a character string.
	COMPRESS Function (p. 375)	Returns a character string with specified characters removed from the original string.
	COUNTC Function (p. 393)	Counts the number of characters in a string that appear or do not appear in a list of characters.
	COUNTW Function (p. 396)	Counts the number of words in a character string.
	CURRENT_LOCALE Function (p. 407)	Returns the five character name of the current locale.
	DEQUOTE Function (p. 423)	Removes matching single quotation marks from a character string that begins with a single quotation mark, and deletes all characters to the right of the closing quotation mark.
	INDEX Function (p. 553)	Searches a character expression for a string of characters, and returns the position of the string's first character for the first occurrence of the string.

Category	Language Elements	Description
	INDEXC Function (p. 555)	Searches a character expression for specified characters and returns the position of the first occurrence of any of the characters.
	LEFT Function (p. 614)	Left aligns a character expression.
	LOWCASE Function (p. 631)	Converts all letters in a character expression to lowercase.
	NOTALNUM Function (p. 665)	Searches a character string for a non-alphanumeric character, and returns the first character position at which the character is found.
	NOTALPHA Function (p. 667)	Searches a character string for a nonalphabetic character, and returns the first character position at which the character is found.
	NOTDIGIT Function (p. 669)	Searches a character string for any character that is not a digit, and returns the first character position at which that character is found.
	NOTFIRST Function (p. 671)	Searches a character string for an invalid first character in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.
	NOTLOWER Function (p. 673)	Searches a character string for a character that is not a lowercase letter, and returns the first character position at which that character is found.
	NOTNAME Function (p. 675)	Searches a character string for an invalid character in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.
	NOTPUNCT Function (p. 677)	Searches a character string for a character that is not a punctuation character, and returns the first character position at which that character is found.
	NOTSPACE Function (p. 679)	Searches a character string for a character that is not a whitespace character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first character position at which that character is found.
	NOTUPPER Function (p. 681)	Searches a character string for a character that is not an uppercase letter, and returns the first character position at which that character is found.
	NOTXDIGIT Function (p. 683)	Searches a character string for a character that is not a hexadecimal character, and returns the first character position at which that character is found.
	OCTET_LENGTH Function (p. 689)	Returns the number of bytes in a string of any data type.
	QUOTE Function (p. 778)	Adds double quotation marks to a character value.

Category	Language Elements	Description
	RANK Function (p. 782)	Returns the position of a character in the ASCII collating sequence.
	REPEAT Function (p. 783)	Repeats a character expression.
	REVERSE Function (p. 784)	Reverses a character expression.
	SCAN Function (p. 795)	Returns the nth word from a character expression.
	SUBSTRING Function (p. 823)	Extracts a substring from a character string.
	TRANSTRN Function (p. 839)	Replaces or removes all occurrences of a substring in a character string.
	TRANWRD Function (p. 841)	Replaces or removes all occurrences of a substring in a character string.
	TRIM Function (p. 844)	Removes leading characters, trailing characters, or both from a character string.
	UPCASE Function (p. 847)	Converts all letters in an argument to uppercase.
	VERIFY Function (p. 855)	Returns the position of the first character that is unique to an expression.
	WHICHC Function (p. 861)	Returns the first position of a character string from a list of character strings.
Combinatorial	COMB Function (p. 367)	Computes the number of combinations of n elements taken r at a time.
	LCOMB Function (p. 613)	Computes the logarithm of the COMB function, which is the logarithm of the number of combinations of n objects taken r at a time.
	LFACT Function (p. 615)	Computes the logarithm of the FACT (factorial) function.
	PERM Function (p. 738)	Computes the number of permutations of n items that are taken r at a time.
Date and Time	CURRENT_DATE Function (p. 406)	Returns the current date for the time zone.
	CURRENT_TIME Function (p. 407)	Returns the current time for your time zone.
	CURRENT_TIME_GMT Function (p. 408)	Returns the current GMT time.
	CURRENT_TIMESTAMP Function (p. 409)	Returns the date and time for your time zone.

Category	Language Elements	Description
	CURRENT_TIMESTAMP_GMT Function (p. 410)	Returns the current GMT date and time.
	DATDIF Function (p. 413)	Returns the number of days between two dates after computing the difference between the dates according to specified day count conventions.
	DATE Function (p. 416)	Returns the current date as a SAS date value.
	DATEJUL Function (p. 417)	Converts a Julian date to a SAS date value.
	DATEPART Function (p. 418)	Returns the date as year, month, and day.
	DATETIME Function (p. 419)	Returns the current date and time of day as a SAS datetime value.
	DAY Function (p. 420)	Returns the numeric day of the month from a date or datetime value.
	DHMS Function (p. 429)	Returns a SAS datetime value from date, hour, minute, and second values.
	HMS Function (p. 544)	Returns a SAS time value from hour, minute, and second values.
	HOLIDAY Function (p. 545)	Returns a SAS date value of a specified holiday for a specified year.
	HOUR Function (p. 549)	Returns the hour from a time or datetime value.
	INTCINDEX Function (p. 561)	Returns the cycle index when a date, time, or timestamp interval and value are specified.
	INTCK Function (p. 564)	Returns the number of interval boundaries of a given kind that lie between two SAS dates, times, or timestamp values encoded as DOUBLE.
	INTCYCLE Function (p. 571)	Returns the date, time, or datetime interval at the next higher seasonal cycle when a date, time, or datetime interval is specified.
	INTFIT Function (p. 574)	Returns a time interval that is aligned between two dates.
	INTGET Function (p. 575)	Returns a time interval based on three date or datetime values.
	INTINDEX Function (p. 578)	Returns the seasonal index when a date, time, or timestamp interval and value are specified.
	INTNX Function (p. 582)	Increments a SAS date, time, or datetime value encoded as a DOUBLE, and returns a SAS date, time, or datetime value encoded as a DOUBLE.

Category	Language Elements	Description
	INTSEAS Function (p. 590)	Returns the length of the seasonal cycle when a date, time, or datetime interval is specified.
	INTSHIFT Function (p. 594)	Returns the shift interval that corresponds to the base interval.
	INTTEST Function (p. 596)	Returns 1 if a time interval is valid, and returns 0 if a time interval is invalid.
	JULDATE Function (p. 606)	Returns the Julian date from a SAS date value.
	JULDATE7 Function (p. 607)	Returns a seven-digit Julian date from a SAS date value.
	MAKEDATE Function (p. 633)	Returns the date as year, month, and day.
	MAKETIME Function (p. 634)	Returns the time as hours, minutes, and seconds.
	MAKETIMESTAMP Function (p. 635)	Returns the timestamp.
	MDY Function (p. 644)	Returns a SAS date value from month, day, and year values.
	MINUTE Function (p. 649)	Returns the minute from a time or datetime value.
	MONTH Function (p. 655)	Returns the numeric month from a date or datetime value.
	NWKDOM Function (p. 686)	Returns the date for the nth occurrence of a weekday for the specified month and year.
	QTR Function (p. 773)	Returns the quarter of the year from a SAS date value.
	SECOND Function (p. 806)	Returns the second from a time or datetime value.
	TIMEPART Function (p. 831)	Returns the time as hours, minutes, and seconds.
	TODAY Function (p. 838)	Returns the current date as a numeric SAS date value.
	WEEK Function (p. 857)	Returns the week-number value.
	WEEKDAY Function (p. 860)	From a SAS date value, returns a whole number that corresponds to the day of the week.
	YEAR Function (p. 864)	Returns the year from a date or datetime value.
	YRDIF Function (p. 867)	Returns the difference in years between two dates according to specified day count conventions; returns a person's age.
	YYQ Function (p. 870)	Returns a SAS date value from year and quarter year values.

Category	Language Elements	Description
Descriptive Statistics	AVG Function (p. 290)	Returns the average of all values in a column.
	CSS Function (p. 401)	Returns the corrected sum of squares of all values in an expression.
	CV Function (p. 411)	Returns the coefficient of variation.
	GEOMEAN Function (p. 537)	Returns the geometric mean.
	GEOMEANZ Function (p. 539)	Returns the geometric mean, using zero fuzzing.
	HARMEAN Function (p. 540)	Returns the harmonic mean.
	HARMEANZ Function (p. 542)	Returns the harmonic mean, using zero fuzzing.
	IQR Function (p. 602)	Returns the interquartile range.
	KURTOSIS Function (p. 609)	Returns the kurtosis of all values in an expression.
	LARGEST Function (p. 610)	Returns the kth largest non-null or nonmissing value.
	MAD Function (p. 632)	Returns the median absolute deviation from the median.
	MAX Function (p. 642)	Returns the maximum value in a column.
	MEAN Function (p. 645)	Returns the arithmetic mean (average) of the non-null or nonmissing arguments.
	MEDIAN Function (p. 646)	Returns the median value.
	MIN Function (p. 648)	Returns the minimum value in an expression.
	ORDINAL Function (p. 690)	Orders a list of values, and returns a value that is based on a position in the list.
	PCTL Function (p. 691)	Returns the percentile that corresponds to the percentage.
	PROBT Function (p. 768)	Returns the probability from a t distribution of the values in an expression.
	RANGE Function (p. 780)	Returns the range between values in an expression.
	RMS Function (p. 785)	Returns the root mean square.
	SKEWNESS Function (p. 811)	Returns the skewness of all values in an expression.
	SMALLEST Function (p. 812)	Returns the kth smallest non-null or nonmissing value.
	STD Function (p. 818)	Returns the standard deviation.

Category	Language Elements	Description
	STDDEV Function (p. 819)	Returns the statistical standard deviation of all values in an expression.
	STDERR Function (p. 820)	Returns the statistical standard error of all values in an expression.
	STUDENTS_T Function (p. 822)	Returns the Student's t distribution of the values in an expression.
	SUM Function (p. 825)	Returns the sum of all the values in an expression.
	SUMABS Function (p. 826)	Returns the sum of the absolute values of the nonmissing arguments.
	USS Function (p. 851)	Returns the uncorrected sum of squares of all the values in an expression.
	VAR Function (p. 853)	Returns the variance.
	VARIANCE Function (p. 854)	Returns the measure of the dispersion of all values in an expression.
Financial	BLACKCLPRC Function (p. 295)	Calculates call prices for European options on futures, based on the Black model.
	BLACKPTPRC Function (p. 297)	Calculates put prices for European options on futures, based on the Black model.
	BLKSHCLPRC Function (p. 299)	Calculates call prices for European options on stocks, based on the Black-Scholes model.
	BLKSHPTPRC Function (p. 302)	Calculates put prices for European options on stocks, based on the Black-Scholes model.
	COMPOUND Function (p. 374)	Returns compound interest parameters.
	CONVX Function (p. 384)	Returns the convexity for an enumerated cash flow.
	CONVXP Function (p. 386)	Returns the convexity for a periodic cash flow stream, such as a bond.
	CUMIPMT Function (p. 403)	Returns the cumulative interest paid on a loan between the start and end period.
	CUMPRINC Function (p. 404)	Returns the cumulative principal paid on a loan between the start and end period.
	DUR Function (p. 432)	Returns the modified duration for an enumerated cash flow.
	DURP Function (p. 434)	Returns the modified duration for a periodic cash flow stream, such as a bond.
	EFFRATE Function (p. 437)	Returns the effective annual interest rate.

Category	Language Elements	Description
	FINANCE Function (p. 443)	Computes financial calculations such as depreciation, maturation, accrued interest, net present value, periodic savings, and internal rates of return.
	FINANCE ACCRINT Function (p. 447)	Computes the accrued interest for a security that pays periodic interest.
	FINANCE ACCRINTM Function (p. 449)	Computes the accrued interest for a security that pays interest at maturity.
	FINANCE AMORDEGRC Function (p. 451)	Computes the depreciation for each accounting period by using a depreciation coefficient.
	FINANCE AMORLINC Function (p. 452)	Computes the depreciation for each accounting period.
	FINANCE COUPDAYBS Function (p. 454)	Computes the number of days from the beginning of the coupon period to the settlement date.
	FINANCE COUPDAYS Function (p. 456)	Computes the number of days in the coupon period that contains the settlement date.
	FINANCE COUPDAYSNC Function (p. 457)	Computes the number of days from the settlement date to the next coupon date.
	FINANCE COUPNCD Function (p. 459)	Computes the next coupon date after the settlement date.
	FINANCE COUPNUM Function (p. 460)	Computes the number of coupons that are payable between the settlement date and the maturity date.
	FINANCE COUPPCD Function (p. 462)	Computes the previous coupon date before the settlement date.
	FINANCE CUMIPMT Function (p. 463)	Computes the cumulative interest paid between two periods.
	FINANCE CUMPRINC Function (p. 465)	Computes the cumulative principal that is paid on a loan between two periods.
	FINANCE DB Function (p. 466)	Computes the depreciation of an asset for a specified period by using the fixed-declining balance method.
	FINANCE DDB Function (p. 467)	Computes the depreciation of an asset for a specified period by using the double-declining balance method or some other method that you specify.
	FINANCE DISC Function (p. 468)	Computes the discount rate for a security.
	FINANCE DOLLARDE Function (p. 470)	Converts a dollar price, expressed as a fraction, to a dollar price, expressed as a decimal number.

Category	Language Elements	Description
	FINANCE DOLLARFR Function (p. 471)	Converts a dollar price, expressed as a decimal number, to a dollar price, expressed as a fraction.
	FINANCE DURATION Function (p. 472)	Computes the annual duration of a security with periodic interest payments.
	FINANCE EFFECT Function (p. 473)	Computes the effective annual interest rate.
	FINANCE FV Function (p. 474)	Computes the future value of an investment.
	FINANCE FVSCHEDULE Function (p. 475)	Computes the future value of the initial principal after applying a series of compound interest rates.
	FINANCE INTRATE Function (p. 476)	Computes the interest rate for a fully invested security.
	FINANCE IPMT Function (p. 478)	Computes the interest payment for an investment for a specified period.
	FINANCE IRR Function (p. 479)	Computes the internal rate of return for a series of cash flows.
	FINANCE ISPMT Function (p. 480)	Calculates the interest paid during a specific period of an investment.
	FINANCE MDURATION Function (p. 481)	Computes the Macaulay modified duration for a security with an assumed face value of $100.
	FINANCE MIRR Function (p. 483)	Computes the internal rate of return where positive and negative cash flows are financed at different rates.
	FINANCE NOMINAL Function (p. 484)	Computes the annual nominal interest rates.
	FINANCE NPER Function (p. 485)	Computes the number of periods for an investment.
	FINANCE NPV Function (p. 486)	Computes the net present value of an investment based on a series of periodic cash flows and a discount rate.
	FINANCE ODDFPRICE Function (p. 487)	Computes the price of a security per $100 face value with an odd first period.
	FINANCE ODDFYIELD Function (p. 489)	Computes the yield of a security with an odd first period.
	FINANCE ODDLPRICE Function (p. 491)	Computes the price of a security per $100 face value with an odd last period.
	FINANCE ODDLYIELD Function (p. 493)	Computes the yield of a security with an odd last period.

Category	Language Elements	Description
	FINANCE PMT Function (p. 495)	Computes the periodic payment of an annuity.
	FINANCE PPMT Function (p. 496)	Computes the payment on the principal for an investment for a specified period.
	FINANCE PRICE Function (p. 497)	Computes the price of a security per $100 face value that pays periodic interest.
	FINANCE PRICEDISC Function (p. 499)	Computes the price of a discounted security per $100 face value.
	FINANCE PRICEMAT Function (p. 500)	Computes the price of a security per $100 face value that pays interest at maturity.
	FINANCE PV Function (p. 502)	Computes the present value of an investment.
	FINANCE RATE Function (p. 503)	Computes the interest rate per period of an annuity.
	FINANCE RECEIVED Function (p. 505)	Computes the amount that is received at maturity for a fully invested security.
	FINANCE SLN Function (p. 506)	Computes the straight-line depreciation of an asset for one period.
	FINANCE SYD Function (p. 507)	Computes the sum-of-years digits depreciation of an asset for a specified period.
	FINANCE TBILLEQ Function (p. 508)	Computes the bond-equivalent yield for a treasury bill.
	FINANCE TBILLPRICE Function (p. 509)	Computes the price of a treasury bill per $100 face value.
	FINANCE TBILLYIELD Function (p. 510)	Computes the yield for a treasury bill.
	FINANCE VDB Function (p. 511)	Computes the depreciation of an asset for a specified or partial period by using a declining balance method.
	FINANCE XIRR Function (p. 513)	Computes the internal rate of return for a schedule of cash flows that is not necessarily periodic.
	FINANCE XNPV Function (p. 514)	Computes the net present value for a schedule of cash flows that is not necessarily periodic.
	FINANCE YIELD Function (p. 515)	Computes the yield on a security that pays periodic interest.
	FINANCE YIELDDISC Function (p. 517)	Computes the annual yield for a discounted security (for example, a treasury bill).

Category	Language Elements	Description
	FINANCE YIELDMAT Function (p. 518)	Computes the annual yield of a security that pays interest at maturity.
	GARKHCLPRC Function (p. 531)	Calculates call prices for European options on stocks, based on the Garman-Kohlhagen model.
	GARKHPTPRC Function (p. 533)	Calculates put prices for European options on stocks, based on the Garman-Kohlhagen model.
	INTRR Function (p. 588)	Returns the internal rate of return as a decimal value.
	IPMT Function (p. 600)	Returns the interest payment for a given period for a constant payment loan or the periodic savings for a future balance.
	IRR Function (p. 603)	Returns the internal rate of return as a percentage.
	MARGRCLPRC Function (p. 636)	Calculates call prices for European options on stocks, based on the Margrabe model.
	MARGRPTPRC Function (p. 639)	Calculates put prices for European options on stocks, based on the Margrabe model.
	MORT Function (p. 656)	Returns amortization parameters.
	NETPV Function (p. 660)	Returns the net present value as a percent.
	NOMRATE Function (p. 664)	Returns the nominal annual interest rate.
	NPV Function (p. 685)	Returns the net present value with the rate expressed as a percentage.
	PMT Function (p. 740)	Returns the periodic payment for a constant payment loan or the periodic savings for a future balance.
	PVP Function (p. 771)	Returns the present value for a periodic cash flow stream (such as a bond), with repayment of principal at maturity.
	SAVING Function (p. 791)	Returns the future value of a periodic saving.
	SAVINGS Function (p. 792)	Returns the balance of a periodic savings by using variable interest rates.
	TIMEVALUE Function (p. 832)	Returns the equivalent of a reference amount at a base date by using variable interest rates.
	YIELDP Function (p. 866)	Returns the yield-to-maturity for a periodic cash flow stream, such as a bond.
Hyperbolic	ARCOSH Function (p. 281)	Returns the inverse hyperbolic cosine.
	ARSINH Function (p. 284)	Returns the inverse hyperbolic sine.
	ARTANH Function (p. 285)	Returns the inverse hyperbolic tangent.

Category	Language Elements	Description
Mathematical	ABS Function (p. 259)	Returns the absolute value of a numeric value expression.
	AIRY Function (p. 260)	Returns the value of the Airy function.
	BETA Function (p. 292)	Returns the value of the beta function.
	CNONCT Function (p. 362)	Returns the noncentrality parameter from a chi-square distribution.
	COALESCE Function (p. 364)	Returns the first non-null or nonmissing value from a list of numeric arguments.
	CONSTANT Function (p. 378)	Computes machine and mathematical constants.
	DAIRY Function (p. 412)	Returns the derivative of the AIRY function.
	DEVIANCE Function (p. 425)	Returns the deviance based on a probability distribution.
	DIGAMMA Function (p. 431)	Returns the value of the digamma function.
	E Function (p. 436)	Returns the natural logarithm, e.
	ERF Function (p. 438)	Returns the value of the (normal) error function.
	ERFC Function (p. 439)	Returns the value of the complementary (normal) error function.
	EXP Function (p. 441)	Returns the value of the e constant raised to a specified power.
	FACT Function (p. 442)	Computes a factorial.
	FNONCT Function (p. 525)	Returns the value of the noncentrality parameter of an F distribution.
	GAMMA Function (p. 530)	Returns the value of the gamma function.
	GCD Function (p. 536)	Returns the greatest common divisor for a set of integers.
	IBESSEL Function (p. 550)	Returns the value of the modified Bessel function.
	JBESSEL Function (p. 605)	Returns the value of the Bessel function.
	LCM Function (p. 612)	Returns the least common multiple for a set of whole numbers.
	LGAMMA Function (p. 616)	Returns the natural logarithm of the Gamma function.
	LOG Function (p. 617)	Returns the natural logarithm (base e) of a numeric value expression.
	LOG1PX Function (p. 618)	Returns the log of 1 plus the argument.

Category	Language Elements	Description
	LOG2 Function (p. 619)	Returns the base-2 logarithm of a numeric value expression.
	LOG10 Function (p. 620)	Returns the base-10 logarithm of a numeric value expression.
	LOGBETA Function (p. 621)	Returns the logarithm of the beta function.
	LOGISTIC Function (p. 625)	Returns the logistic transformation of the argument.
	MOD Function (p. 651)	Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid most unexpected floating-point results.
	MODZ Function (p. 653)	Returns the remainder from the division of the first argument by the second argument, using zero fuzzing.
	PI Function (p. 739)	Returns the constant value of PI as a floating-point value.
	POWER Function (p. 743)	Returns the value of a numeric value expression raised to a specified power.
	SIGN Function (p. 807)	Returns a number that indicates the sign of a numeric value expression.
	SQRT Function (p. 817)	Returns the square root of a value.
	TNONCT Function (p. 836)	Returns the value of the noncentrality parameter from the Student's t distribution.
	TRIGAMMA Function (p. 843)	Returns the value of the trigamma function.
	WHICHN Function (p. 863)	Returns the first position of a number from a list of numbers.
Probability	CDF Function (p. 312)	Computes the left cumulative distribution function from various continuous and discrete probability distributions.
	CDF BERNOULLI Distribution Function (p. 315)	Returns a value from the Bernoulli cumulative probability distribution.
	CDF BETA Distribution Function (p. 316)	Returns a value from the beta cumulative probability distribution.
	CDF BINOMIAL Distribution Function (p. 318)	Returns a value from the binomial cumulative probability distribution.
	CDF CAUCHY Distribution Function (p. 320)	Returns a value from the Cauchy cumulative probability distribution.
	CDF Chi-Square Distribution Function (p. 322)	Returns a value from the chi-square cumulative probability distribution.

Category	Language Elements	Description
	CDF Conway-Maxwell-Poisson Distribution Function (p. 324)	Returns a value from the Conway-Maxwell-Poisson cumulative probability distribution.
	CDF Exponential Distribution Function (p. 325)	Returns a value from the exponential cumulative probability distribution.
	CDF F Distribution Function (p. 327)	Returns a value from the F cumulative probability distribution.
	CDF GAMMA Distribution Function (p. 329)	Returns a value from the gamma cumulative probability distribution.
	CDF Generalized Poisson Distribution Function (p. 330)	Returns a value from the generalized Poisson cumulative probability distribution.
	CDF GEOMETRIC Distribution Function (p. 332)	Returns a value from the geometric cumulative probability distribution.
	CDF HYPERGEOMETRIC Distribution Function (p. 333)	Returns a value from the hypergeometric cumulative probability distribution.
	CDF LAPLACE Distribution Function (p. 335)	Returns a value from the Laplace cumulative probability distribution.
	CDF LOGISTIC Distribution Function (p. 337)	Returns a value from the logistic cumulative probability distribution.
	CDF LOGNORMAL Distribution Function (p. 339)	Returns a value from the lognormal cumulative probability distribution.
	CDF NEGBINOMIAL Distribution Function (p. 340)	Returns a value from the negative binomial cumulative probability distribution.
	CDF NORMAL Distribution Function (p. 342)	Returns a value from the normal cumulative probability distribution.
	CDF NORMALMIX Distribution Function (p. 344)	Returns a value from the normal mixture cumulative probability distribution.
	CDF PARETO Distribution Function (p. 346)	Returns a value from the Pareto cumulative probability distribution.
	CDF POISSON Distribution Function (p. 347)	Returns a value from the Poisson cumulative probability distribution.
	CDF T Distribution Function (p. 349)	Returns a value from the T cumulative probability distribution.
	CDF TWEEDIE Distribution Function (p. 351)	Returns a value from the Tweedie cumulative probability distribution.
	CDF UNIFORM Distribution Function (p. 353)	Returns a value from the uniform cumulative probability distribution.

Category	Language Elements	Description
	CDF WALD (Inverse Gaussian) Distribution Function (p. 354)	Returns a value from the Wald (also known as the inverse Gaussian) cumulative probability distribution.
	CDF WEIBULL Distribution Function (p. 356)	Returns a value from the Weibull cumulative probability distribution.
	LOGCDF Function (p. 622)	Returns the logarithm of a left cumulative distribution function.
	LOGPDF Function (p. 626)	Computes the logarithm of the probability density (mass) function from various continuous and discrete distributions.
	LOGSDF Function (p. 628)	Returns the logarithm of a survival function.
	PDF Function (p. 693)	Returns a value from a probability density (mass) distribution.
	PDF BERNOULLI Distribution Function (p. 695)	Returns a value from the Bernoulli probability density (mass) distribution.
	PDF BETA Distribution Function (p. 697)	Returns a value from the beta probability density (mass) distribution.
	PDF BINOMIAL Distribution Function (p. 699)	Returns a value from the binomial probability density (mass) distribution.
	PDF CAUCHY Distribution Function (p. 700)	Returns a value from the Cauchy probability density (mass) distribution.
	PDF Chi-Square Distribution Function (p. 702)	Returns a value from the chi-square probability density (mass) distribution.
	PDF Conway-Maxwell-Poisson Distribution Function (p. 704)	Returns a value from the Conway-Maxwell-Poisson probability density (mass) distribution.
	PDF EXPONENTIAL Distribution Function (p. 706)	Returns a value from the exponential probability density (mass) distribution.
	PDF F Distribution Function (p. 708)	Returns a value from the F probability density (mass) distribution.
	PDF GAMMA Distribution Function (p. 710)	Returns a value from the gamma probability density (mass) distribution.
	PDF Generalized Poisson Distribution Function (p. 711)	Returns a value from the generalized Poisson probability density (mass) distribution.
	PDF GEOMETRIC Distribution Function (p. 713)	Returns a value from the geometric probability density (mass) distribution.
	PDF Hypergeometric Distribution Function (p. 714)	Returns a value from a hypergeometric probability density (mass) distribution.

Category	Language Elements	Description
	PDF LAPLACE Distribution Function (p. 716)	Returns a value from the Laplace probability density (mass) distribution.
	PDF LOGISTIC Distribution Function (p. 718)	Returns a value from the logistic probability density (mass) distribution.
	PDF LOGNORMAL Distribution Function (p. 720)	Returns a value from the lognormal probability density (mass) distribution.
	PDF NEGBINOMIAL Distribution Function (p. 721)	Returns the value from the negative binomial probability density (mass) distribution.
	PDF NORMAL Distribution Function (p. 723)	Returns a value from the normal probability density (mass) distribution.
	PDF NORMALMIX Distribution Function (p. 724)	Returns a value from the normal mixture probability density (mass) distribution.
	PDF PARETO Distribution Function (p. 726)	Returns a value from the Pareto probability density (mass) distribution.
	PDF POISSON Distribution Function (p. 728)	Returns a value from the Poisson probability density (mass) distribution.
	PDF T Distribution Function (p. 729)	Returns a value from the T probability density (mass) distribution.
	PDF TWEEDIE Distribution Function (p. 731)	Returns a value from the Tweedie probability density (mass) distribution.
	PDF UNIFORM Distribution Function (p. 733)	Returns a value from the uniform probability density (mass) distribution.
	PDF Wald (Inverse Gaussian) Distribution Function (p. 735)	Returns a value from the Wald (also known as the inverse Gaussian) probability density (mass) distribution.
	PDF WEIBULL Distribution Function (p. 736)	Returns a value from the Weibull probability density (mass) distribution.
	POISSON Function (p. 742)	Returns the probability from a Poisson distribution.
	PROBBETA Function (p. 744)	Returns the probability from a beta distribution.
	PROBBNML Function (p. 745)	Returns the probability from a binomial distribution.
	PROBBNRM Function (p. 746)	Returns a probability from a bivariate normal distribution.
	PROBCHI Function (p. 748)	Returns the probability from a chi-square distribution.
	PROBF Function (p. 749)	Returns the probability from an F distribution.

Category	Language Elements	Description
	PROBGAM Function (p. 750)	Returns the probability from a gamma distribution.
	PROBHYPR Function (p. 751)	Returns the probability from a hypergeometric distribution.
	PROBMC Function (p. 754)	Returns a probability or a quantile from various distributions for multiple comparisons of means.
	PROBMED Function (p. 764)	Computes cumulative probabilities for the sample median.
	PROBNEGB Function (p. 765)	Returns the probability from a negative binomial distribution.
	PROBNORM Function (p. 767)	Returns the probability from the standard normal distribution.
	SDF Function (p. 801)	Returns a survival function.
Quantile	BETAINV Function (p. 293)	Returns a quantile from the beta distribution.
	GAMINV Function (p. 528)	Returns a quantile from the gamma distribution.
	PROBIT Function (p. 753)	Returns a quantile from the standard normal distribution.
	QUANTILE Function (p. 774)	Returns the quantile from a distribution when you specify the left probability (CDF).
	SQUANTILE Function (p. 814)	Returns the quantile from a distribution when you specify the right probability (SDF).
	TINV Function (p. 834)	Returns a quantile from the t distribution.
Scalar	NMISS Function (p. 661)	Returns the number of null values or SAS missing values in an expression.
Special	CAST Function (p. 310)	Converts a value from one data type to another.
	FMTINFO Function (p. 523)	Returns information about a SAS format or informat.
	IFNULL Function (p. 551)	Checks the value of the first expression and, if it is null or a SAS missing value, returns the second expression.
	INPUTC Function (p. 556)	Enables you to specify a character informat at run time.
	INPUTN Function (p. 558)	Enables you to specify a numeric informat at run time.
	N Function (p. 658)	Returns the number of non-null or nonmissing numeric values.
	PUT Function (p. 769)	Returns a value using a specified format.
	SYSGET Function (p. 827)	Returns the value of the specified operating environment variable.

Category	Language Elements	Description
Trigonometric	ARCOS Function (p. 280)	Returns the arccosine in radians.
	ARSIN Function (p. 283)	Returns the arcsine in radians.
	ATAN Function (p. 287)	Returns the arctangent in radians.
	ATAN2 Function (p. 288)	Returns the arctangent of the x and y coordinates of a right triangle, in radians.
	COS Function (p. 388)	Returns the cosine in radians.
	COSH Function (p. 389)	Returns the hyperbolic cosine in radians.
	COT Function (p. 390)	Returns the tangent in radians.
	CSC Function (p. 400)	Returns the cosecant.
	DEGREES Function (p. 421)	Returns the number of degrees for an angle in radians.
	RADIANS Function (p. 779)	Returns the number of radians converted from a numeric degree value.
	SEC Function (p. 804)	Returns the secant.
	SIN Function (p. 808)	Returns the trigonometric sine.
	SINH Function (p. 809)	Returns the hyperbolic sine.
	TAN Function (p. 829)	Returns the tangent.
	TANH Function (p. 830)	Returns the hyperbolic tangent.
Truncation	CEIL Function (p. 358)	Returns the smallest integer greater than or equal to a numeric value expression.
	CEILZ Function (p. 359)	Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing.
	FLOOR Function (p. 520)	Returns the largest integer less than or equal to a numeric value expression.
	FLOORZ Function (p. 521)	Returns the largest integer that is less than or equal to the argument, using zero fuzzing.
	FUZZ Function (p. 527)	Returns the nearest whole number if the argument is within 1E-12 of that number.
	INT Function (p. 559)	Returns the whole number, fuzzed to avoid unexpected floating-point results.
	INTZ Function (p. 598)	Returns the whole number portion of the argument, using zero fuzzing.

Category	Language Elements	Description
	ROUND Function (p. 786)	Rounds the first argument to the nearest multiple of the second argument, or to the nearest integer when the second argument is omitted.
	ROUNDZ Function (p. 789)	Rounds the first argument to the nearest multiple of the second argument, using zero fuzzing.
	TRUNC Function (p. 846)	Truncates a numeric value to a specified length.
Web Tools	URLDECODE Function (p. 848)	Returns a string that was decoded using the URL escape syntax.
	URLENCODE Function (p. 850)	Returns a string that was encoded using the URL escape syntax.

Dictionary

ABS Function

Returns the absolute value of a numeric value expression.

Categories: Mathematical
CAS

Returned data type: The same data type as the expression

Syntax

ABS(*expression*)

Arguments

expression
specifies any valid SQL expression that evaluates to a numeric value.

Type BIGINT, DOUBLE, FLOAT, INTEGER, REAL, SMALLINT, TINYINT

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

Details

If *expression* is null, then the ABS function returns null. If the result is a number that does not fit into the range of the argument's data type, the ABS function fails.

Example

The following statements illustrate the ABS function:

Statements	Results
`select abs(-345);`	`345`
`select abs((3 * 50) / 5)`	`30`

AIRY Function

Returns the value of the Airy function.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

AIRY(*x*)

Arguments

x
specifies a numeric constant, variable, or expression.

Data type DOUBLE

Details

The AIRY function returns the value of the Airy function. (See a list of References.) It is the solution of the differential equation

$$w^{(2)} - xw = 0$$

with the conditions

$$w(0) = \frac{1}{3^{\frac{2}{3}}\Gamma\left(\frac{2}{3}\right)}$$

and

$$w'(0) = -\frac{1}{3^{\frac{1}{3}}\Gamma\left(\frac{1}{3}\right)}$$

Example

The following statements illustrate the AIRY function:

Statements	Results
`select airy(2.0);`	`0.034924`
`select airy(-2.0);`	`0.227407`

ANYALNUM Function

Searches a character string for an alphanumeric character, and returns the first character position at which the character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

ANYALNUM('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYALNUM function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYALNUM function searches a string for the first occurrence of any character that is a digit or an uppercase or lowercase letter. If such a character is found, ANYALNUM returns the position in the string of that character. If no such character is found, ANYALNUM returns a value of 0.

If you use only one argument, ANYALNUM begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYALNUM returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYALNUM function searches a character string for an alphanumeric character. The NOTALNUM function searches a character string for a non-alphanumeric character.

Example

The following statements illustrate the ANYALNUM function:

Statements	Results
`select anyalnum('abc123, x');`	`1 (position of a)`
`select anyalnum('+abcd');`	`2 (position of a)`
`select anyalnum('789ab');`	`1 (position of 7)`
`select anyalnum('!&@start');`	`4 (position of s)`
`select anyalnum('!&@start1',-99);`	`9 (position of last alphanumeric in string)`

See Also

Functions:

- "NOTALNUM Function" on page 665

ANYALPHA Function

Searches a character string for an alphabetic character, and returns the first character position at which the character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

ANYALPHA('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYALPHA function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYALPHA function searches a string for the first occurrence of any character that is an uppercase or lowercase letter. If such a character is found, ANYALPHA returns the position in the string of that character. If no such character is found, ANYALPHA returns a value of 0.

If you use only one argument, ANYALPHA begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYALPHA returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYALPHA function searches a character string for an alphabetic character. The NOTALPHA function searches a character string for a non-alphabetic character.

Example

The following statements illustrate the ANYALPHA function:

Statements	Results
`select anyalpha('123abc456');`	`4 (position of a)`

Statements	Results
`select anyalpha('abcd1234');`	`1 (position of a)`
`select anyalpha('12345');`	`0 (no letters)`
`select anyalpha('789abc456',-99);`	`6 (position of last letter in string)`

See Also

Functions:

- "NOTALPHA Function" on page 667

ANYDIGIT Function

Searches a character string for a digit, and returns the first character position at which the digit is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

ANYDIGIT('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The ANYDIGIT function does not depend on the TRANTAB, ENCODING, or LOCALE system options.

The ANYDIGIT function searches a string for the first occurrence of any character that is a digit. If such a character is found, ANYDIGIT returns the position in the string of that character. If no such character is found, ANYDIGIT returns a value of 0.

If you use only one argument, ANYDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYDIGIT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYDIGIT function searches a character string for a digit. The NOTDIGIT function searches a character string for any character that is not a digit.

Example

The following statements illustrate the ANYDIGIT function:

Statements	Results
`select anydigit('123abc456');`	`1 (position of 1)`
`select anydigit('abcd456);`	`5 (position of 4)`
`select anydigit('abcde');`	`0 (no digits)`
`select anydigit('789ab',-99);`	`3 (position of last digit in string)`

See Also

Functions:

- “NOTDIGIT Function” on page 669

ANYFIRST Function

Searches a character string for a character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

ANYFIRST('*expression*'[, *start*])

Arguments

expression

specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

start

specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The ANYFIRST function does not depend on the TRANTAB, ENCODING, or LOCALE system options.

The ANYFIRST function searches a string for the first occurrence of any character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7. These characters are the underscore (_) and uppercase or lowercase English letters. If such a character is found, ANYFIRST returns the position in the string of that character. If no such character is found, ANYFIRST returns a value of 0.

If you use only one argument, ANYFIRST begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYFIRST returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYFIRST function searches a string for the first occurrence of any character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7. The NOTFIRST function searches a string for the first occurrence of any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7.

Example

The following statements illustrate the ANYFIRST function:

Statements	Results
`select anyfirst('123aBc4D6');`	`4 (position of a)`
`select anyfirst('Bac123, x');`	`1 (position of B)`
`select anyfirst('!?:;,');`	`0`
`select anyfirst('1,2,3,x');`	`7 (position of x)`
`select anyfirst('x,1,2,3');`	`1 (position of x)`
`select anyfirst('x,1,2,3',-99);`	`1 (position of last letter)`

Statements	Results
`select anyfirst('&,x,y,z,20,!');`	`3 (position of x)`
`select anyfirst('&,x,y,z,20,!',-99);`	`7 (position of last letter in string)`

See Also

Functions:

- "NOTFIRST Function" on page 671

ANYLOWER Function

Searches a character string for a lowercase letter, and returns the first character position at which the letter is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

ANYLOWER('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYLOWER function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYLOWER function searches a string for the first occurrence of a lowercase letter. If such a character is found, ANYLOWER returns the position in the string of that character. If no such character is found, ANYLOWER returns a value of 0.

If you use only one argument, ANYLOWER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYLOWER returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYLOWER function searches a character string for a lowercase letter. The NOTLOWER function searches a character string for a character that is not a lowercase letter.

Example

The following statements illustrate the ANYLOWER function:

Statements	Results
`select anylower('123aBc4D6');`	`4 (position of a)`
`select anylower('123aBc4D6',-99);`	`6 (position of last lowercase letter)`
`select anylower('DAN BROWN');`	`0 (no lowercase letters)`

See Also

Functions:

- “NOTLOWER Function” on page 673

ANYNAME Function

Searches a character string for a character that is valid in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

ANYNAME('*expression*'[,*start*])

Arguments

expression

specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

start

specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The ANYNAME function does not depend on the TRANTAB, ENCODING, or LOCALE system options.

The ANYNAME function searches a string for the first occurrence of any character that is valid in a SAS variable name under VALIDVARNAME=V7. These characters

are the underscore (_), digits, and uppercase or lowercase English letters. If such a character is found, ANYNAME returns the position in the string of that character. If no such character is found, ANYNAME returns a value of 0.

If you use only one argument, ANYNAME begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYNAME returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYNAME function searches a string for the first occurrence of any character that is valid in a SAS variable name under VALIDVARNAME=V7. The NOTNAME function searches a string for the first occurrence of any character that is not valid in a SAS variable name under VALIDVARNAME=V7.

Example

The following statements illustrate the ANYNAME function:

Statements	Results
`select anyname('<&abc!');`	`3 (position of a)`
`select anyname('&abc!');`	`2 (position of a)`
`select anyname('abc!>');`	`1 (position of a)`
`select anyname('789!>');`	`1 (position of 7)`
`select anyname('abc!>', -99);`	`3 (position of c)`

See Also

Functions:

- “NOTNAME Function” on page 675

ANYPUNCT Function

Searches a character string for a punctuation character, and returns the first character position at which that character is found.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

ANYPUNCT('*expression*'[, *start*])

Arguments

expression

specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

start

specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYPUNCT function depend directly on the translation table that is in effect (see “TRANTAB= System Option” in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYPUNCT function searches a string for the first occurrence of a punctuation character. If such a character is found, ANYPUNCT returns the position in the string of that character. If no such character is found, ANYPUNCT returns a value of 0.

If you use only one argument, ANYPUNCT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYPUNCT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYPUNCT function searches a character string for a punctuation character. The NOTPUNCT function searches a character string for a character that is not a punctuation character.

Example

The following statements illustrate the ANYPUNCT function:

Statements	Results
`select anypunct('abc123, x');`	`7 (position of comma)`
`select anypunct('!?:;,');`	`1 (position of exclamation mark)`
`select anypunct('!?:;,', -99);`	`5 (position of last punctuation mark in string)`
`select anypunct('987&efg');`	`4 (position of ampersand)`

See Also

Functions:

- “NOTPUNCT Function” on page 677

ANYSPACE Function

Searches a character string for a whitespace character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first character position at which that character is found.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

ANYSPACE('*expression*'[, *start*])

Arguments

expression

specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

start

specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYSPACE function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYSPACE function searches a string for the first occurrence of any character that is a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. If such a character is found, ANYSPACE returns the position in the string of that character. If no such character is found, ANYSPACE returns a value of 0.

If you use only one argument, ANYSPACE begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*,

specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYSPACE returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYSPACE function searches a character string for the first occurrence of a character that is a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. The NOTSPACE function searches a character string for the first occurrence of a character that is not a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed.

Example

The following statements illustrate the ANYSPACE function:

Statements	Results
`select anyspace('abc123, x');`	`8 (position of space before x)`
`select anyspace('Ma ry Smith',-99);`	`6 (position of last space in string)`

See Also

Functions:

- "NOTSPACE Function" on page 679

ANYUPPER Function

Searches a character string for an uppercase letter, and returns the first character position at which the letter is found.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

ANYUPPER('*expression*'[, *start*])

Arguments

expression

specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

start

specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYUPPER function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYUPPER function searches a string for the first occurrence of an uppercase letter. If such a character is found, ANYUPPER returns the position in the string of that character. If no such character is found, ANYUPPER returns a value of 0.

If you use only one argument, ANYUPPER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYUPPER returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYUPPER function searches a character string for an uppercase letter. The NOTUPPER function searches a character string for a character that is not an uppercase letter.

Example

The following statements illustrate the ANYUPPER function:

Statements	Results
`select anyupper('123aBc4D6');`	`5 (position of first uppercase letter)`
`select anyupper('123aBc4D6',-99);`	`8 (position of last uppercase letter in string)`
`select anyupper('abcde');`	`0 (no uppercase letters)`

See Also

Functions:

- “NOTUPPER Function” on page 681

ANYXDIGIT Function

Searches a character string for a hexadecimal character that represents a digit, and returns the first character position at which that character is found.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

ANYXDIGIT('*expression*'[,*start*])

Arguments

expression

specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

start

specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The ANYXDIGIT function does not depend on the TRANTAB, ENCODING, or LOCALE system options.

The ANYXDIGIT function searches a string for the first occurrence of any character that is a digit or an uppercase or lowercase A, B, C, D, E, or F. If such a character is found, ANYXDIGIT returns the position in the string of that character. If no such character is found, ANYXDIGIT returns a value of 0.

If you use only one argument, ANYXDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYXDIGIT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYXDIGIT function searches a character string for a character that is a hexadecimal character. The NOTXDIGIT function searches a character string for a character that is not a hexadecimal character.

Example

The following statements illustrate the ANYXDIGIT function:

Statements	Results
`select anyxdigit('123aBc4D6');`	`1 (position of 1)`
`select anyxdigit('abcd456');`	`1 (position of a)`
`select anyxdigit('789ab!',-99);`	`5 (position of b)`
`select anyxdigit('&789ab!');`	`2 (position of 7)`

See Also

Functions:

- "NOTXDIGIT Function" on page 683

ARCOS Function

Returns the arccosine in radians.

Categories:	Trigonometric CAS
Alias:	ACOS
Returned data type:	DOUBLE

Syntax

ARCOS(*expression*)

Arguments

expression

specifies any valid SQL expression that evaluates to a numeric value.

Range -1 to 1

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The ARCOS function returns the arccosine (inverse cosine) of the argument. The value that is returned is specified in radians.

Example

The following statements illustrate the ARCOS function:

Statements	Results
`select arcos(1);`	`0`
`select arcos(0);`	`1.570796`
`select arcos(-0.5);`	`2.094395`

See Also

Functions:

- "ARSIN Function" on page 283
- "COS Function" on page 388
- "SIN Function" on page 808

ARCOSH Function

Returns the inverse hyperbolic cosine.

Categories: CAS

	Hyperbolic
Returned data type:	DOUBLE

Syntax

ARCOSH(*expression*)

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Range	*expression* >= 1
Data type	DOUBLE
See	"<sql-expression> Expression" on page 894
	"FedSQL Expressions" on page 46

Details

The ARCOSH function computes the inverse hyperbolic cosine. The ARCOSH function is mathematically defined by the following equation, where *expression* >= 1. In the equation, *expression* is represented by *x*.

$$ARCOSH(x) = log\left(x + \sqrt{x^2 - 1}\right)$$

Example

The following statements illustrate the ARCOSH function:

Statements	Results
`select arcosh(5);`	`2.292432`
`select arcosh(13);`	`3.256614`

See Also

Functions:

- “ARSINH Function” on page 284
- “ARTANH Function” on page 285
- “COSH Function” on page 389
- “SINH Function” on page 809
- “TANH Function” on page 830

ARSIN Function

Returns the arcsine in radians.

Categories:	Trigonometric CAS
Alias:	ASIN
Returned data type:	DOUBLE

Syntax

ARSIN(*expression*)

Arguments

expression

specifies any valid SQL expression that evaluates to a numeric value.

Range	–1 to 1
Data type	DOUBLE
See	“<sql-expression> Expression” on page 894 “FedSQL Expressions” on page 46

Example

The following statements illustrate the ARSIN function:

Statements	Results
`select arsin(0);`	`0`
`select arsin(1);`	`1.570796`

Statements	Results
`select arsin(-0.5);`	`-0.5236`

See Also

Functions:

- “ARCOS Function” on page 280
- “COS Function” on page 388
- “COSH Function” on page 389
- “SIN Function” on page 808

ARSINH Function

Returns the inverse hyperbolic sine.

Categories:	CAS Hyperbolic
Returned data type:	DOUBLE

Syntax

ARSINH(*expression*)

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Range	$-\infty < x < \infty$
Data type	DOUBLE
See	“<sql-expression> Expression” on page 894 “FedSQL Expressions” on page 46

Details

The ARSINH function computes the inverse hyperbolic sine. The ARSINH function is mathematically defined by the following equation, where $-\infty < x < \infty$.

$$ARSINH(x) = log\left(x + \sqrt{x^2 + 1}\right)$$

Replace the infinity symbol with the largest double precision number that is available on your machine. In the equation, *expression* is represented by *x*.

Example

The following statements illustrate the ARSINH function:

Statements	Results
`select arsinh(5);`	`2.312438`
`select arsinh(-5);`	`-2.31244`

See Also

Functions:

- "ARSINH Function" on page 284
- "ARTANH Function" on page 285
- "COSH Function" on page 389
- "SINH Function" on page 809
- "TANH Function" on page 830

ARTANH Function

Returns the inverse hyperbolic tangent.

Categories:	CAS Hyperbolic
Returned data type:	DOUBLE

Syntax

ARTANH(*expression*)

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Range –1 < *expression* < 1

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The ARTANH function computes the inverse hyperbolic tangent. The ARTANH function is mathematically defined by the following equation, where –1 < *expression*< 1. In the equation, *expression* is represented by *x*.

$$ARTANH(x) = \frac{1}{2}log\left(\frac{1+x}{1-x}\right)$$

Example

The following statements illustrate the ARTANH function:

Statements	Results
`select artanh(.5);`	`0.549306`
`select artanh(-.5);`	`-0.54931`

See Also

Functions:

- "ARSINH Function" on page 284
- "ARTANH Function" on page 285
- "COSH Function" on page 389
- "SINH Function" on page 809

■ "TANH Function" on page 830

ATAN Function

Returns the arctangent in radians.

Categories:	Trigonometric CAS
Alias:	ARTAN
Returned data type:	DOUBLE

Syntax

ATAN(*expression*)

Arguments

expression

specifies any valid SQL expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The ATAN function returns the 2-quadrant arctangent (inverse tangent) of the argument. The value that is returned is the angle (in radians) whose tangent is *x* and whose value ranges from - $\pi/2$ to $\pi/2$. If the argument is missing, then ATAN returns a missing value.

Comparisons

The ATAN function is similar to the ATAN2 function except that ATAN2 calculates the arc tangent of the angle from the values of two arguments rather than from one argument.

Example

The following statements illustrate the ATAN function:

Statements	Results
`select atan(0);`	`0`
`select atan(1);`	`0.785398`
`select atan(-9.0);`	`-1.46014`

See Also

Functions:

- “ATAN2 Function” on page 288
- “COT Function” on page 390
- “TAN Function” on page 829
- “TANH Function” on page 830

ATAN2 Function

Returns the arctangent of the *x* and *y* coordinates of a right triangle, in radians.

Categories:	Trigonometric CAS
Returned data type:	DOUBLE

Syntax

ATAN2(*expression-1*, *expression-2*)

Arguments

expression-1

specifies any valid SQL expression that evaluates to a numeric value. *expression-1* specifies the x coordinate of the end of the hypotenuse of a right triangle.

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

expression-2
specifies any valid SQL expression that evaluates to a numeric value. *expression-2* specifies the y coordinate of the end of the hypotenuse of a right triangle.

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

Details

The ATAN2 function returns the arctangent (inverse tangent) of two numeric variables. The result of this function is similar to the result of calculating the arctangent of *expression-1* / *expression-2*, except that the signs of both arguments are used to determine the quadrant of the result. If either of the arguments in ATAN2 is missing, then ATAN2 returns either a null or a SAS missing value.

Comparisons

The ATAN2 function is similar to the ATAN function except that ATAN calculates the arctangent of the angle from the value of one argument rather than from two arguments.

Example

The following statements illustrate the ATAN2 function:

Statements	Results
`select atan2(-1, 0.5);`	`-1.10715`
`select atan2(6,8);`	`0.643501`
`select atan2(5,-3);`	`2.111216`

See Also

- “How FedSQL Processes Nulls and SAS Missing Values” on page 20

Functions:

- "ATAN Function" on page 287
- "TAN Function" on page 829
- "TANH Function" on page 830

AVG Function

Returns the average of all values in a column.

Categories:	Aggregate Descriptive Statistics CAS
Alias:	MEAN
Returned data type:	DOUBLE

Syntax

AVG([DISTINCT] *expression*)

Arguments

expression
specifies any valid SQL expression.

Data type BIGINT, DOUBLE, FLOAT, INTEGER, REAL, SMALLINT, TINYINT

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

DISTINCT
(optional) eliminates duplicate rows from the aggregate calculation.

Details

The AVG function adds the values of all the rows in the specified column and divides the result by the number of rows. Null values and SAS missing values are ignored and are not included in the computation.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the

data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Example

Table: DENSITIES on page 1173

The following statements illustrate the AVG function:

Statements	Results
`select avg(density) from densities;`	`172.8324`
`select avg(population) from densities;`	`12277544`

See Also

Functions:

- "SUM Function" on page 825

Clauses:

- "SELECT Clause" on page 956
- "GROUP BY Clause" on page 968
- "HAVING Clause" on page 970

BAND Function

Returns the bitwise logical AND of two arguments.

Categories: Bitwise Logical Operations
CAS

Returned data type: DOUBLE

Syntax

BAND(*expression-1, expression-2*)

Arguments

expression-1, expression-2
specifies any valid expression that evaluates to a numeric value.

Range between 0 and (2^{32})–1 inclusive

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Example

The following statements illustrate the BAND function:

Statements	Results
`select band(9,11);`	9
`select band(15,5);`	5

BETA Function

Returns the value of the beta function.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

BETA(*a*, *b*)

Arguments

a
is the first shape parameter.

Range *a* >0

Data type DOUBLE

b

is the second shape parameter.

Range $b > 0$

Data type DOUBLE

Details

The BETA function is mathematically given by this equation:

$$\beta(a,b) = \int_0^1 x^{a-1}(1-x)^{b-1}dx$$

Note the following:

$$\beta(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

In the previous equation, $\Gamma(.)$ is the gamma function.

If the expression cannot be computed, BETA returns a missing value.

Example

The following statements illustrate the BETA function:

Statements	Results
`select beta(5,3);`	`0.00952380952381`
`select beta(15,45);`	`1.6710294365E-15`

See Also

Functions:

- "LOGBETA Function" on page 621

BETAINV Function

Returns a quantile from the beta distribution.

Categories: CAS
Quantile

Returned data type: DOUBLE

Syntax

BETAINV(*p*, *a*, *b*)

Arguments

p

is a numeric probability.

Range $0 \le p \le 1$

Data type DOUBLE

a

is a numeric shape parameter.

Range $a > 0$

Data type DOUBLE

b

is a numeric shape parameter.

Range $b > 0$

Data type DOUBLE

Details

The BETAINV function returns the *p*th quantile from the beta distribution with shape parameters *a* and *b*. The probability that an observation from a beta distribution is less than or equal to the returned quantile is *p*.

Note: BETAINV is the inverse of the PROBBETA function.

Example

The following statement illustrates the BETAINV function:

Statements	Results
`select betainv(0.001,2,4);`	`0.0101017879`

See Also

Functions:

- "PROBBETA Function" on page 744

BLACKCLPRC Function

Calculates call prices for European options on futures, based on the Black model.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

BLACKCLPRC(*E*, *t*, *F*, *r*, *sigma*)

Arguments

E

is a nonmissing, positive value that specifies exercise price.

Requirement Specify *E* and *F* in the same units.

Data type DOUBLE

t

is a nonmissing value that specifies time to maturity, in years.

Data type DOUBLE

F

is a nonmissing, positive value that specifies future price.

Requirement Specify *F* and *E* in the same units.

Data type DOUBLE

r

is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

Data type DOUBLE

sigma

is a nonmissing, positive fraction that specifies the volatility (the square root of the variance of *r*).

Data type DOUBLE

Details

The BLACKCLPRC function calculates call prices for European options on futures, based on the Black model. The function is based on the following relationship:

$$\text{CALL} = \varepsilon^{-rt}(FN(d_1) - EN(d_2))$$

Arguments

F
specifies future price.

N
specifies the cumulative normal density function.

E
specifies the exercise price of the option.

r
specifies the risk-free interest rate. This is an annual rate that is expressed in terms of continuous compounding.

t
specifies the time to expiration, in years.

$$d_1 = \frac{\left(\ln\left(\frac{F}{E}\right) + \left(\frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

The following arguments apply to the preceding equation:

σ
specifies the volatility of the underlying asset.

σ^2
specifies the variance of the rate of return.

For the special case of *t*=0, the following equation is true:

$$\text{CALL} = \max((F - E), 0)$$

For information about the basics of pricing, see "Using Pricing Functions" in *SAS Functions and CALL Routines: Reference*.

Comparisons

The BLACKCLPRC function calculates call prices for European options on futures, based on the Black model. The BLACKPTPRC function calculates put prices for European options on futures, based on the Black model. These functions return a scalar value.

Example

The following statements illustrate the BLACKCLPRC function:

Statements	Results
	`----+----1----+---2--`
`select blackclprc(50,.25,48,.05,.25);`	`1.55130142723117`
`select blackclprc(9,1/12,10,.05,.2);`	`1`

See Also

Functions:

- "BLACKPTPRC Function" on page 297

BLACKPTPRC Function

Calculates put prices for European options on futures, based on the Black model.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

BLACKPTPRC(*E*, *t*, *F*, *r*, *sigma*)

Arguments

E

is a nonmissing, positive value that specifies exercise price.

Requirement Specify *E* and *F* in the same units.

Data type DOUBLE

t

is a nonmissing value that specifies time to maturity, in years.

Data type DOUBLE

F
is a nonmissing, positive value that specifies future price.

Requirement Specify *F* and *E* in the same units.

Data type DOUBLE

r
is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

Data type DOUBLE

sigma
is a nonmissing, positive fraction that specifies the volatility (the square root of the variance of *r*).

Data type DOUBLE

Details

The BLACKPTPRC function calculates put prices for European options on futures, based on the Black model. The function is based on the following relationship:

$$\mathrm{PUT} = \mathrm{CALL} + \varepsilon^{-rt}(E - F)$$

Arguments

E
specifies the exercise price of the option.

r
specifies the risk-free interest rate. This is an annual rate that is expressed in terms of continuous compounding.

t
specifies the time to expiration, in years.

F
specifies future price.

$$d_1 = \frac{\left(\ln\left(\frac{F}{E}\right) + \left(\frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

The following arguments apply to the preceding equation:

σ
specifies the volatility of the underlying asset.

σ^2
specifies the variance of the rate of return.

For the special case of *t*=0, the following equation is true:

$$\mathrm{PUT} = \max((E - F), 0)$$

For information about the basics of pricing, see "Using Pricing Functions" in *SAS Functions and CALL Routines: Reference*.

Comparisons

The BLACKPTPRC function calculates put prices for European options on futures, based on the Black model. The BLACKCLPRC function calculates call prices for European options on futures, based on the Black model. These functions return a scalar value.

Example

The following statements illustrate the BLACKPTPRC function:

Statements	Results
	`----+----1----+---2--`
`select blackptprc(298,.25,350,.06,.25);`	`1.85980563934967`
`select blackptprc(145,.5,170,.05,.2);`	`1.41234979911583`

See Also

Functions:

- "BLACKCLPRC Function" on page 295

BLKSHCLPRC Function

Calculates call prices for European options on stocks, based on the Black-Scholes model.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

BLKSHCLPRC(*E*, *t*, *S*, *r*, *sigma*)

Arguments

E

is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

Data type DOUBLE

t

is a nonmissing value that specifies the time to maturity, in years.

Data type DOUBLE

S

is a nonmissing, positive value that specifies the share price.

Requirement Specify *S* and *E* in the same units.

Data type DOUBLE

r

is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

Data type DOUBLE

sigma

is a nonmissing, positive fraction that specifies the volatility of the underlying asset.

Data type DOUBLE

Details

The BLKSHCLPRC function calculates the call prices for European options on stocks, based on the Black-Scholes model. The function is based on the following relationship:

$$\text{CALL} = SN(d_1) - EN(d_2)\varepsilon^{-rt}$$

Arguments

S

is a nonmissing, positive value that specifies the share price.

N

specifies the cumulative normal density function.

E

is a nonmissing, positive value that specifies the exercise price of the option.

$$d_1 = \frac{\left(\ln\left(\frac{S}{E}\right) + \left(r + \frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

The following arguments apply to the preceding equation:

t
specifies the time to expiration, in years.

r
specifies the risk-free interest rate. This is an annual rate that is expressed in terms of continuous compounding.

σ
specifies the volatility (the square root of the variance).

σ^2
specifies the variance of the rate of return.

For the special case of *t*=0, the following equation is true:

$$\mathrm{CALL} = \max((S - E), 0)$$

For information about the basics of pricing, see "Using Pricing Functions" in *SAS Functions and CALL Routines: Reference*.

Comparisons

The BLKSHCLPRC function calculates the call prices for European options on stocks, based on the Black-Scholes model. The BLKSHPTPRC function calculates the put prices for European options on stocks, based on the Black-Scholes model. These functions return a scalar value.

Example

The following statements illustrate the BLKSHCLPRC function:

Statements	Results
	`----+----1----+---2--`
`select blkshclprc(50,.25,48,.05,.25);`	`1.79894201954463`
`select blkshclprc(9,1/12,10,.05,.2);`	`1`

See Also

Functions:

- "BLKSHPTPRC Function" on page 302

BLKSHPTPRC Function

Calculates put prices for European options on stocks, based on the Black-Scholes model.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

BLKSHPTPRC(*E*, *t*, *S*, *r*, *sigma*)

Arguments

E

is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

Data type DOUBLE

t

is a nonmissing value that specifies the time to maturity, in years.

Data type DOUBLE

S

is a nonmissing, positive value that specifies the share price.

Requirement Specify *S* and *E* in the same units.

Data type DOUBLE

r

is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

Data type DOUBLE

sigma

is a nonmissing, positive fraction that specifies the volatility of the underlying asset.

Data type DOUBLE

Details

The BLKSHPTPRC function calculates the put prices for European options on stocks, based on the Black-Scholes model. The function is based on the following relationship:

$$\text{PUT} = \text{CALL} - S + E\varepsilon^{-rt}$$

Arguments

S
is a nonmissing, positive value that specifies the share price.

E
is a nonmissing, positive value that specifies the exercise price of the option.

$$d_1 = \frac{\left(\ln\left(\frac{S}{E}\right) + \left(r + \frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

The following arguments apply to the preceding equation:

t
specifies the time to expiration, in years.

r
specifies the risk-free interest rate. This is an annual rate that is expressed in terms of continuous compounding.

σ
specifies the volatility (the square root of the variance).

σ^2
specifies the variance of the rate of return.

For the special case of *t*=0, the following equation is true:

$$\text{PUT} = \max((E - S), 0)$$

For information about the basics of pricing, see "Using Pricing Functions" in *SAS Functions and CALL Routines: Reference*.

Comparisons

The BLKSHPTPRC function calculates the put prices for European options on stocks, based on the Black-Scholes model. The BLKSHCLPRC function calculates the call prices for European options on stocks, based on the Black-Scholes model. These functions return a scalar value.

Example

The following statements illustrate the BLKSHPTPRC function:

Statements	Results
	`----+----1----+---2--`
`select blkshptprc(230,.5,290,.04,.25);`	`1.56597442946068`
`select blkshptprc(350,.3,400,.05,.2);`	`1.64091943067592`

See Also

Functions:

- "BLKSHCLPRC Function" on page 299

BLSHIFT Function

Returns the bitwise logical left shift of two arguments.

Categories: Bitwise Logical Operations
CAS

Returned data type: DOUBLE

Syntax

BLSHIFT(*expression-1*, *expression-2*)

Arguments

expression-1
specifies any valid expression that evaluates to a numeric value.

Range between 0 and (2^{32})–1 inclusive

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

expression-2
specifies any valid expression that evaluates to a numeric value.

Range 0 to 31, inclusive

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Example

The following statements illustrate the BLSHIFT function:

Statements	Results
`select blshift(7,2);`	`28`

See Also

Functions:

- "BRSHIFT Function" on page 307

BNOT Function

Returns the bitwise logical NOT of an argument.

Categories: Bitwise Logical Operations
CAS

Returned data type: DOUBLE

Syntax

BNOT(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Range between 0 and (2^{32})–1 inclusive

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Example

The following statement illustrates the BNOT function:

Statements	Results
`select bnot(16);`	`4.295E9`
`select put(bnot(16), best16.);`	`4294967279`

BOR Function

Returns the bitwise logical OR of two arguments.

Categories: Bitwise Logical Operations
CAS

Returned data type: DOUBLE

Syntax

BOR(*expression-1, expression-2*)

Arguments

expression-1, expression-2
specifies any valid expression that evaluates to a numeric value.

Range between 0 and (2^{32})–1 inclusive

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Example

The following statements illustrate the BOR function:

Statements	Results
`select bor(4,8);`	`12`

BRSHIFT Function

Returns the bitwise logical right shift of two arguments.

Categories: Bitwise Logical Operations
CAS

Returned data type: DOUBLE

Syntax

BRSHIFT(*expression-1*, *expression-2*)

Arguments

expression-1
specifies any valid expression that evaluates to a numeric value.

Range between 0 and (2^{32})–1 inclusive

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

expression-2
specifies any valid expression that evaluates to a numeric value.

Range 0 to 31, inclusive

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Example

The following statement illustrates the BRSHIFT function:

Statements	Results
`select brshift(64,2);`	`16`

See Also

Functions:

- "BLSHIFT Function" on page 304

BXOR Function

Returns the bitwise logical EXCLUSIVE OR of two arguments.

Categories: Bitwise Logical Operations
CAS

Returned data type: DOUBLE

Syntax

BXOR(*expression-1, expression-2*)

Arguments

expression-1, expression-2
specifies any valid expression that evaluates to a numeric value.

Range between 0 and (2^{32})–1 inclusive

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Example

The following statement illustrates the BXOR function:

Statements	Results
`select bxor(128,64);`	`192`

BYTE Function

Returns one character in the ASCII or the EBCDIC collating sequence.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

BYTE(*n*)

Arguments

n

specifies a whole number that represents a specific ASCII or EBCDIC character.

Range 0–255

Data type DOUBLE

Details

For EBCDIC collating sequences, *n* is between 0 and 255. For ASCII collating sequences, the characters that correspond to values between 0 and 127 represent the standard character set. Other ASCII characters that correspond to values between 128 and 255 are available on certain ASCII operating environments, but the information those characters represent varies with the operating environment.

Example

The following statement illustrates the BYTE function:

Statements	Results	
	ASCII	EBCDIC
	`----+----1----+----2`	`----+----1----+----2`
`select byte(80);`	`P`	`&`

See Also

Functions:

- "RANK Function" on page 782

CAST Function

Converts a value from one data type to another.

Category:	Special
Alias:	The operator "::" can be used instead of the CAST keyword.
Returned data type:	The target data type.

Syntax

CAST(*expression* AS *data-type* [(*length*)])

Arguments

expression
specifies any valid SQL expression.

Data type Expression can resolve to any FedSQL data type supported by the data source.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Appendix 2, “Data Type Reference,” on page 1109

data-type

Is the target data type. See “Details” for information about supported data type conversions.

Restriction The target data type must be supported by FedSQL and by the data source.

See “Data Types” on page 14

Appendix 2, “Data Type Reference,” on page 1109

length

Is an optional integer that specifies the length of the target data type. The length argument is intended for use with character values. It is important when specifying literals. When a literal value is specified, the default value is 0, and the length needs to be at least as long as the number of characters that will be generated by the CAST.

Restriction The *length* argument is not available for VARCHAR in the UTF-8 character set. A cast to VARCHAR(*n*) in the UTF-8 character set yields an output value containing N*4 bytes and as many characters as will fit in those bytes.

Details

FedSQL performs implicit type conversions as needed for data source support. The CAST function is provided for performing explicit type conversions.

CAST expressions are permitted anywhere expressions are permitted. In addition to converting one data type to another, CAST can be used to change the length of the target type.

The following type conversions are supported:

- A character type can be converted to another character type, a numeric type, and a DATE or TIMESTAMP.
- A numeric type can be converted to CHAR, VARCHAR, or another numeric type. If the target type cannot represent the non-fractional component without truncation, an exception is raised. If the target numeric cannot represent the fractional component (scale) of the source numeric, then the source is silently truncated to fit into the target. For example, casting 123.4763 as INTEGER yields 123.
- A DATE or TIME value can be converted to a TIMESTAMP and a DOUBLE. TIMESTAMP can also be converted to a DOUBLE. If a DATE is converted to a TIMESTAMP, the TIME component of the resulting TIMESTAMP is always 00:00:00. If a TIME data value is converted to a TIMESTAMP, the DATE component is set to the value of CURRENT_DATE at the time the CAST is executed. If a TIMESTAMP is converted to a DATE, the TIME component is silently truncated. If a TIMESTAMP is converted to a TIME, the DATE component is silently truncated.

Comparisons

The CAST function permanently modifies the data type of the specified input variable. The PUT function affects the output of the query in which it is specified.

Example

Table: "CustonLine" on page 1172

Table: "Integers" on page 1177

The following statements illustrate the CAST function.

Statements	Results
`select cast(begintime as DATE) from custonline;`	01SEP2013 02OCT2013 15OCT2013 01NOV2013 01DEC2013 02JAN2013 16JAN2013 01FEB2013 01MAR2013 15MAR2013
`select si::integer as int from integers;`	32767
`select bi::bigint * 2::bigint as bigbang from integers;`	ERROR: Numeric value out of range
`select cast ('2014-04-10' as DATE);`	10APR2014
`select cast ('2014-04-10 10:56:49' as TIMESTAMP);`	10APR2014:10:56:49
`select cast ('10:56:49' as TIME);`	10:56:49

CDF Function

Computes the left cumulative distribution function from various continuous and discrete probability distributions.

Categories: CAS
Probability

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 774.

Syntax

CDF(*'distribution'*, *quantile* [, *parameter-1, …, parameter-k*])

Arguments

distribution

is a character constant, variable, or expression that identifies the distribution. Here are valid distributions:

Distribution	Argument
Bernoulli	'BERNOULLI'
Beta	'BETA'
Binomial	'BINOMIAL'
Cauchy	'CAUCHY'
Chi-Square	'CHISQUARE'
Conway-Maxwell-Poisson	'CONMAXPOI'
Exponential	'EXPONENTIAL'
F	'F'
Gamma	'GAMMA'
Generalized Poisson	'GENPOISSON'
Geometric	'GEOMETRIC'
Hypergeometric	'HYPERGEOMETRIC'
Laplace	'LAPLACE'
Logistic	'LOGISTIC'
Lognormal	'LOGNORMAL'
Negative binomial	'NEGBINOMIAL'
Normal	'NORMAL'|'GAUSS'
Normal mixture	'NORMALMIX'
Pareto	'PARETO'

Distribution	Argument
Poisson	'POISSON'
T	'T'
Tweedie	'TWEEDIE'
Uniform	'UNIFORM'
Wald (inverse Gaussian)	'WALD'|'IGAUSS'
Weibull	'WEIBULL'

Note Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

quantile
is a numeric constant, variable, or expression that specifies the value of the random variable.

Data type DOUBLE

parameter-1, … , parameter-k
are optional constants, variables, or expressions that specify the values of *shape*, *location*, or *scale* parameters that are appropriate for the specific distribution.

Data type DOUBLE

See Also

Functions:

- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626
- "LOGSDF Function" on page 628
- "PDF Function" on page 693
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

CDF BERNOULLI Distribution Function

Returns a value from the Bernoulli cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 774.

Syntax

CDF('BERNOULLI', *x*, *p*)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

p

is a numeric constant, variable, or expression that specifies a probability of success.

Range $0 \le p \le 1$

Data type DOUBLE

Details

The CDF function for the Bernoulli distribution returns the probability that an observation from a Bernoulli distribution, with probability of success equal to *p*, is less than or equal to *x*.

$$CDF('BERN', x, p) = \begin{cases} 0 & x < 0 \\ 1 - p & 0 \le x < 1 \\ 1 & x \ge 1 \end{cases}$$

Note: There are no *location* or *scale* parameters for this distribution.

Example

The following statement illustrates the CDF Bernoulli distribution function:

Statements	Results
`select cdf('BERN', 0, .25);`	`0.75`

See Also

Functions:

- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “PDF BERNOULLI Distribution Function” on page 695
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

CDF BETA Distribution Function

Returns a value from the beta cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 774.

Syntax

CDF('BETA', *x*, *a*, *b* [, *l*, *r*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $a > 0$

Data type DOUBLE

b

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $b > 0$

Data type DOUBLE

l

is a numeric constant, variable, or expression that specifies the left location parameter.

Default 0

Data type DOUBLE

r

is a numeric constant, variable, or expression that specifies the right location parameter.

Default 1

Range $r > l$

Data type DOUBLE

Details

The CDF function for the beta distribution returns the probability that an observation from a beta distribution, with shape parameters *a* and *b*, is less than or equal to *v*. The following equation describes the CDF function of the beta distribution:

$$CDF('BETA', x, a, b, l, r) = \begin{cases} 0 & x \le l \\ \frac{1}{\beta(a,b)} \int_l^x \frac{(v-l)^{a-1}(r-v)^{b-1}}{(r-l)^{a+b-1}} dv & l < x \le r \\ 1 & x > r \end{cases}$$

The following relationship applies to the preceding equation:

$$\beta(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

The following relationship applies to the preceding equation:

$$\Gamma(a) = \int_0^\infty x^{a-1} e^{-x} dx$$

Example

The following statement illustrates the CDF Beta distribution function:

Statements	Results
`select cdf('BETA', 0.2, 3, 4);`	`0.09888`

See Also

Functions:

- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626
- "LOGSDF Function" on page 628
- "PDF BETA Distribution Function" on page 697
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

CDF BINOMIAL Distribution Function

Returns a value from the binomial cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 774.

Syntax

CDF('BINOMIAL', *m*, *p*, *n*)

Arguments

m

is a whole number, random variable that counts the number of successes.

Range *m* = 0, 1, ...

Data type DOUBLE

p

is a numeric constant, variable, or expression that specifies a probability of success parameter.

Range 0≤*p* ≤1

Data type DOUBLE

n

is a numeric constant, variable, or expression that specifies a whole number parameter that counts the number of independent Bernoulli trials.

Range *n* = 0, 1, ...

Data type DOUBLE

Details

The CDF function for the binomial distribution returns the probability that an observation from a binomial distribution, with parameters *p* and *n*, is less than or equal to *m*.

$$CDF('BINOM', m, p, n) = \begin{cases} 0 & m < 0 \\ \sum_{j=0}^{m} \binom{n}{j} p^j (1-p)^{n-j} & 0 \le m \le n \\ 1 & m > n \end{cases}$$

Note: There are no *location* or *scale* parameters for the binomial distribution.

Example

The following statement illustrates the CDF Binomial distribution function:

Statements	Results
`select cdf('BINOM', 4, .5, 10);`	`0.376953`

See Also

Functions:

- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “PDF BINOMIAL Distribution Function” on page 699
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

CDF CAUCHY Distribution Function

Returns a value from the Cauchy cumulative probability distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 774.

Syntax

CDF('CAUCHY', *x* [,*θ*] [, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The CDF function for the Cauchy distribution returns the probability that an observation from a Cauchy distribution, with the location parameter θ and the scale parameter λ, is less than or equal to *x*.

$$CDF('CAUCHY', x, \theta, \lambda) = \frac{1}{2} + \frac{1}{\pi} tan^{-1}\left(\frac{x-\theta}{\lambda}\right)$$

Example

The following statement illustrates the CDF Cauchy distribution function:

Statements	Results
`select cdf('CAUCHY', 2);`	`0.852416`

See Also

Functions:

- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626
- "LOGSDF Function" on page 628
- "PDF CAUCHY Distribution Function" on page 700
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

CDF Chi-Square Distribution Function

Returns a value from the chi-square cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 774.

Syntax

CDF('CHISQUARE', *x*, *df* [, *nc*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

df

is a numeric constant, variable, or expression that specifies a degrees of freedom parameter.

Range *df* > 0

Data type DOUBLE

nc

is a numeric constant, variable, or expression that specifies an optional noncentrality parameter.

Range *nc* ≥ 0

Data type DOUBLE

Details

The CDF function for the chi-square distribution returns the probability that an observation from a chi-square distribution, with *df* degrees of freedom and the noncentrality parameter *nc*, is less than or equal to *x*. This function accepts non-integer degrees of freedom. If *nc* is omitted or equal to zero, the value returned is from the central chi-square distribution. In the following equation, let $\nu = df$ and let

$\lambda = nc$. The following equation describes the CDF function of the chi-square distribution:

$$CDF('CHISQ', x, \nu, \lambda) = \begin{cases} 0 & x < 0 \\ \sum_{j=0}^{\infty} e^{-\frac{\lambda}{2}} \frac{\left(\frac{\lambda}{2}\right)^j}{j!} P_c(x, \nu + 2j) & x \geq 0 \end{cases}$$

In the equation, $P_c(.,.)$ denotes the probability from the central chi-square distribution:

$$P_c(x, a) = P_g\left(\frac{x}{2}, \frac{a}{2}\right)$$

In the equation, $P_g(y,b)$ is the probability from the gamma distribution given by the equation:

$$P_g(y, b) = \frac{1}{\Gamma(b)} \int_0^y e^{-v} v^{b-1} dv$$

Example

The following statement illustrates the CDF Chi-Square distribution function:

Statements	Results
`select cdf('CHISQ', 11.264, 11);`	`0.578581`

See Also

Functions:

- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “PDF Chi-Square Distribution Function” on page 702
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

CDF Conway-Maxwell-Poisson Distribution Function

Returns a value from the Conway-Maxwell-Poisson cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 774.

Syntax

CDF('CONMAXPOI', *y*, *λ*,*v*)

Arguments

y

is a numeric constant, variable, or expression that specifies a nonnegative whole number that represents counts data.

Data type DOUBLE

λ

is similar to the mean, as in the Poisson distribution.

Data type DOUBLE

v

is a numeric constant, variable, or expression that specifies a dispersion parameter.

Data type DOUBLE

Details

The CDF function returns cumulative probability from 0 to y. For more information, see "Conway-Maxwell-Poisson" distribution in the PDF function on page 704.

Example

The following statement illustrates the CDF Conway-Maxwell-Poisson distribution function:

Statements	Results
`select cdf('CONMAXPOI', 5, 2.3, .4);`	`0.244541`

See Also

Functions:

- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “PDF Conway-Maxwell-Poisson Distribution Function” on page 704
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

CDF Exponential Distribution Function

Returns a value from the exponential cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 774.

Syntax

CDF('EXPONENTIAL', *x* [,*λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The CDF function for the exponential distribution returns the probability that an observation from an exponential distribution, with the scale parameter λ, is less than or equal to *x*.

$$CDF('EXPO', x, \lambda) = \begin{cases} 0 & x < 0 \\ 1 - e^{-\frac{x}{\lambda}} & x \geq 0 \end{cases}$$

Example

The following statement illustrates the CDF Exponential distribution function:

Statements	Results
`select cdf('EXPO', 1);`	`0.632121`

See Also

Functions:

- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “PDF EXPONENTIAL Distribution Function” on page 706
- “QUANTILE Function” on page 774
- “SDF Function” on page 801

■ "SQUANTILE Function" on page 814

CDF F Distribution Function

Returns a value from the F cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "CDF F Distribution Function" on page 327.

Syntax

CDF('F', *x*, *ndf*, *ddf* [, *nc*])

Arguments

x
is a numeric constant, variable, or expression that specifies a random variable.

ndf
is a numeric constant, variable, or expression that specifies a numerator degrees of freedom parameter.

Range *ndf* > 0

Data type DOUBLE

ddf
is a numeric constant, variable, or expression that specifies a denominator degrees of freedom parameter.

Range *ddf* > 0

Data type DOUBLE

nc
is a numeric constant, variable, or expression that specifies a noncentrality parameter.

Range *nc* ≥ 0

Data type DOUBLE

Details

The CDF function for the *F* distribution returns the probability that an observation from an *F* distribution, with *ndf* numerator degrees of freedom, *ddf* denominator degrees of freedom, and the noncentrality parameter *nc*, is less than or equal to *x*. This function accepts noninteger degrees of freedom for *ndf* and *ddf*. If *nc* is omitted or equal to zero, the value returned is from a central *F* distribution. In the following equation, let $\nu_1 = ndf$, let $\nu_2 = ddf$, and let $\lambda = nc$. The following equation describes the CDF function of the *F* distribution:

$$CDF('F', x, \nu_1, \nu_2, \lambda) = \begin{cases} 0 & x < 0 \\ \sum_{j=0}^{\infty} e^{-\frac{\lambda}{2}} \frac{\left(\frac{\lambda}{2}\right)^j}{j!} P_F(x, \nu_1 + 2j, \nu_2) & x \geq 0 \end{cases}$$

In the equation, $P_f(f,u_1,u_2)$ is the probability from the central *F* distribution with

$$P_F(x, u_1, u_2) = P_B\left(\frac{u_1 x}{u_1 x + u_2}, \frac{u_1}{2}, \frac{u_2}{2}\right)$$

and $P_B(x,a,b)$ is the probability from the standard beta distribution.

Note: There are no *location* or *scale* parameters for the *F* distribution.

Example

The following statement illustrates the CDF F distribution function:

Statements	Results
`select cdf('F', 3.32, 2, 3);`	`0.826393`

See Also

Functions:

- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “PDF F Distribution Function” on page 708
- “QUANTILE Function” on page 774
- “SDF Function” on page 801

■ "SQUANTILE Function" on page 814

CDF GAMMA Distribution Function

Returns a value from the gamma cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 774.

Syntax

CDF('GAMMA', *x*, *a* [, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $a > 0$

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range $\lambda > 0$

Data type DOUBLE

Details

The CDF function for the gamma distribution returns the probability that an observation from a gamma distribution, with the shape parameter *a* and the scale parameter λ, is less than or equal to *x*.

$$CDF('GAMMA', x, a, \lambda) = \begin{cases} 0 & x < 0 \\ \frac{1}{\lambda^a \Gamma(a)} \int_0^x v^{a-1} e^{-\frac{v}{\lambda}} dv & x \geq 0 \end{cases}$$

Example

The following statement illustrates the CDF Gamma distribution function:

Statements	Results
`select cdf('GAMMA', 1, 3);`	`0.080301`

See Also

Functions:

- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “PDF GAMMA Distribution Function” on page 710
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

CDF Generalized Poisson Distribution Function

Returns a value from the generalized Poisson cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 774.

Syntax

CDF(‘GENPOISSON’, *x*, *θ*, *η*)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable. This argument must be a whole number.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a shape parameter.

Range ≤5 and >0

Data type DOUBLE

η

is a numeric constant, variable, or expression that specifies a shape parameter.

Range ≥0 and <0.95

Data type DOUBLE

Tip When η =0, the distribution is the Poisson distribution with a mean and variance of θ. When η>0, the mean is $\theta \div (1-\eta)$ and the variance is $\theta \div (1-\eta)^3$.

Details

The probability mass function for the generalized Poisson distribution follows:

$$f(x;\theta,\eta) = \theta(\theta+\eta x)^{x-1}e^{-\theta-\eta x}/x!, \quad x = 0, 1, 2, ..., \qquad \theta > 0,\ 0 \le \eta < 1$$

If η =0, then the generalized Poisson distribution becomes the standard Poisson distribution with the shape parameter θ.

Example

The following statement illustrates the CDF Generalized Poisson distribution function:

Statements	Results
`select cdf('GENPOISSON', 9, 1, .7);`	`0.906163`

See Also

Functions:

- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626
- "LOGSDF Function" on page 628
- "PDF Generalized Poisson Distribution Function" on page 711
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

CDF GEOMETRIC Distribution Function

Returns a value from the geometric cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 774.

Syntax

CDF('GEOMETRIC', *m*, *p*)

Arguments

m

is a numeric random variable that specifies the number of failures.

Data type	DOUBLE
Tip	Decimal values are rounded down if they are far away from a whole number.

p

is a numeric constant, variable, or expression that specifies a probability of success.

Range	$0 \le p \le 1$
Data type	DOUBLE

Details

The CDF function for the geometric distribution returns the probability that an observation from a geometric distribution, with the parameter *p*, is less than or equal to *m*.

$$CDF('GEOM', m, p) = \begin{cases} 0 & m < 0 \\ 1 - (1 - p)^{(m+1)} & m \geq 0 \end{cases}$$

Note: There are no *location* or *scale* parameters for this distribution.

Example

The following statement illustrates the CDF Geometric distribution function:

Statements	Results
`select cdf('GEOMETRIC', 5, .35);`	`0.924581109375`

See Also

Functions:

- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626
- "LOGSDF Function" on page 628
- "PDF GEOMETRIC Distribution Function" on page 713
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

CDF HYPERGEOMETRIC Distribution Function

Returns a value from the hypergeometric cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 774.

Syntax

CDF('HYPER', *x*, *N*, *R*, *n* [, *o*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable. This argument must be a whole number.

Data type DOUBLE

N

is a numeric constant, variable, or expression that specifies a population size parameter. This argument must be a whole number.

Range $N = 1, 2, \ldots$

Data type DOUBLE

R

is a numeric constant, variable, or expression that specifies a number of items in the category of interest. This argument must be a whole number.

Range $R = 0, 1, \ldots, N$

Data type DOUBLE

n

is a numeric constant, variable, or expression that specifies a sample size parameter. This argument must be a whole number.

Range $n = 1, 2, \ldots, N$

Data type DOUBLE

o

is a numeric constant, variable, or expression that specifies an optional numeric odds ratio parameter.

Range $o > 0$

Data type DOUBLE

Details

The CDF function for the hypergeometric distribution returns the probability that an observation from an extended hypergeometric distribution, with population size *N*, number of items *R*, sample size *n*, and odds ratio *o*, is less than or equal to *x*. If *o* is omitted or equal to 1, the value returned is from the usual hypergeometric distribution.

$$CDF\ (\text{'}HYPER\text{'}, x, N, R, n, o) = \begin{cases} 0 & x < max(0, R+n-N) \\ \dfrac{\sum_{i=0}^{x} \binom{R}{i}\binom{N-R}{n-i} o^i}{\sum_{j=max(0,R+n-N)}^{min(R,n)} \binom{R}{j}\binom{N-R}{n-j} o^j} & max(0, R+n-N) \le x \le min(R,n) \\ 1 & x > min(R,n) \end{cases}$$

Example

The following statement illustrates the CDF Hypergeometric distribution function:

Statements	Results
`select cdf('HYPER', 2, 200, 50, 10);`	`0.523673`

See Also

Functions:

- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “PDF Hypergeometric Distribution Function” on page 714
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

CDF LAPLACE Distribution Function

Returns a value from the Laplace cumulative probability distribution.

Categories: CAS

Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 774.

Syntax

CDF('LAPLACE', *x* [,*θ*, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The CDF function for the Laplace distribution returns the probability that an observation from the Laplace distribution, with the location parameter θ and the scale parameter λ, is less than or equal to *x*.

$$CDF('LAPLACE', x, \theta, \lambda) = \begin{cases} \frac{1}{2}e^{\frac{(x-\theta)}{\lambda}} & x < \theta \\ 1 - \frac{1}{2}e^{\left(-\frac{(x-\theta)}{\lambda}\right)} & x \geq \theta \end{cases}$$

Example

The following statement illustrates the CDF Laplace distribution function:

Statements	Results
`select cdf('LAPLACE', 1);`	`0.81606`

See Also

Functions:

- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “PDF LAPLACE Distribution Function” on page 716
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

CDF LOGISTIC Distribution Function

Returns a value from the logistic cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 774.

Syntax

CDF('LOGISTIC', *x* [, *θ*, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The CDF function for the Logistic distribution returns the probability that an observation from a Logistic distribution, with the location parameter θ and the scale parameter λ, is less than or equal to *x*.

$$CDF('LOGISTIC', x, \theta, \lambda) = \frac{1}{1 + e^{\left(-\frac{x-\theta}{\lambda}\right)}}$$

Example

The following statement illustrates the CDF Logistic distribution function:

Statements	Results
`select cdf('LOGISTIC', 1);`	`0.731059`

See Also

Functions:

- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626

- "LOGSDF Function" on page 628
- "PDF LOGISTIC Distribution Function" on page 718
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

CDF LOGNORMAL Distribution Function

Returns a value from the lognormal cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 774.

Syntax

CDF('LOGNORMAL', *x* [, *θ*, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a log scale parameter. e(θ) is a scale parameter.

Default 0

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a shape parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The CDF function for the lognormal distribution returns the probability that an observation from a lognormal distribution, with the log scale parameter θ and the shape parameter λ, is less than or equal to *x*.

$$CDF('LOGN', x, \theta, \lambda) = \begin{cases} 0 & x \le 0 \\ \frac{1}{\lambda\sqrt{2\pi}} \int_{-\infty}^{log(x)} e\left(-\frac{(v-\theta)^2}{2\lambda^2}\right) dv & x > 0 \end{cases}$$

Example

The following statement illustrates the CDF Lognormal distribution function:

Statements	Results
`select cdf('LOGNORMAL', 1);`	`0.5`

See Also

Functions:

- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “PDF LOGNORMAL Distribution Function” on page 720
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

CDF NEGBINOMIAL Distribution Function

Returns a value from the negative binomial cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 774.

Syntax

CDF('NEGBINOMIAL', *m*, *p*, *n*)

Arguments

m

is a numeric constant, variable, or expression that specifies a random variable that counts the number of failures. This argument must be a positive whole number.

Range *m* = 0, 1, ...

Data type DOUBLE

p

is a numeric constant, variable, or expression that specifies a probability of success.

Range $0 \le p \le 1$

Data type DOUBLE

n

is a numeric constant, variable, or expression that specifies a value that counts the number of successes.

Range $n > 0$

Data type DOUBLE

Details

The CDF function for the negative binomial distribution returns the probability that an observation from a negative binomial distribution, with the probability of success *p* and the number of successes *n*, is less than or equal to *m*.

$$CDF('NEGB', m, p, n) = \begin{cases} 0 & m < 0 \\ p^n \sum_{j=0}^{m} \binom{n+j-1}{n-1} (1-p)^j & m \ge 0 \end{cases}$$

Note: There are no *location* or *scale* parameters for the negative binomial distribution.

Example

The following statement illustrates the CDF Negative Binomial distribution function:

Statements	Results
`select cdf('NEGB', 1, .5, 2);`	`0.5`

See Also

Functions:

- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “PDF NEGBINOMIAL Distribution Function” on page 721
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

CDF NORMAL Distribution Function

Returns a value from the normal cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 774.

Syntax

CDF('NORMAL', x [, θ, λ])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The CDF function for the Normal distribution returns the probability that an observation from the Normal distribution, with the location parameter θ and the scale parameter λ, is less than or equal to *x*.

$$CDF('NORMAL', x, \theta, \lambda) = \frac{1}{\lambda\sqrt{2\pi}} \int_{-\infty}^{x} e\left(-\frac{(v-\theta)^2}{2\lambda^2}\right) dv$$

Example

The following statement illustrates the CDF Normal distribution function:

Statements	Results
`select cdf('NORMAL', 1.96);`	`0.975002`

See Also

Functions:

- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626

- "LOGSDF Function" on page 628
- "PDF NORMAL Distribution Function" on page 723
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

CDF NORMALMIX Distribution Function

Returns a value from the normal mixture cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 774.

Syntax

CDF('NORMALMIX', *x*, *n*, *p*, *m*, *s*)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

n

is a numeric constant, variable, or expression that specifies the number of mixtures.

Range *n* = 1, 2, ...

Data type DOUBLE

p

is a numeric constant, variable, or expression that specifies the *n* proportions, $p_1, p_2, \ldots, p_n$, where $\sum_{i=1}^{i=n} p_i = 1$.

Range *p* = 0, 1, ...

Data type DOUBLE

m

is a numeric constant, variable, or expression that specifies the *n* means $m_1, m_2, \ldots, m_n$.

Data type DOUBLE

s

is a numeric constant, variable, or expression that specifies the *n* standard deviations $s_1, s_2, \ldots, s_n$.

Range $s > 0$

Data type DOUBLE

Details

The CDF function for the Normal Mixture distribution returns the probability that an observation from a mixture of normal distribution is less than or equal to *x*.

$$CDF('NORMALMIX', x, n, p, m, s) = \sum_{i=1}^{i=n} p_i \, CDF('NORMAL', x, m_i, s_i)$$

Weights for the Normal Mixture distribution must be nonnegative. If the sum of the weights does not equal 1, then the weights are treated as relative weights and adjusted so that the sum equals 1.

Note: There are no *location* or *scale* parameters for the Normal Mixture distribution.

Example

The following statement illustrates the CDF Normal Mixture distribution function:

Statements	Results
`select cdf('NORMALMIX',2.3, 3,.33, .33, .34, .5, 1.5, 2.5, .79, 1.6, 4.3);`	`0.718131`

See Also

Functions:

- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628

- "PDF NORMALMIX Distribution Function" on page 724
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

CDF PARETO Distribution Function

Returns a value from the Pareto cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 774.

Syntax

CDF('PARETO', *x*, *a* [, *k*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $a > 0$

Data type DOUBLE

k

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range $k > 0$

Data type DOUBLE

Details

The CDF function for the Pareto distribution returns the probability that an observation from a Pareto distribution, with the shape parameter *a* and the scale parameter *k*, is less than or equal to *x*.

$$CDF('PARETO', x, a, k) = \begin{cases} 0 & x < k \\ 1 - \left(\frac{k}{x}\right)^a & x \geq k \end{cases}$$

Example

The following statement illustrates the CDF Pareto distribution function:

Statements	Results
`select cdf('PARETO', 1 ,1);`	`0`

See Also

Functions:

- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626
- "LOGSDF Function" on page 628
- "PDF PARETO Distribution Function" on page 726
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

CDF POISSON Distribution Function

Returns a value from the Poisson cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 774.

Syntax

CDF('POISSON', *n*, *m*)

Arguments

n

is a numeric constant, variable, or expression that specifies a random variable. This argument must be a whole number.

Range *n* = 0, 1, ...

Data type DOUBLE

m

is a numeric constant, variable, or expression that specifies a mean parameter.

Range *m* > 0

Data type DOUBLE

Details

The CDF function for the Poisson distribution returns the probability that an observation from a Poisson distribution, with mean *m*, is less than or equal to *n*.

$$CDF('POISSON', n, m) = \begin{cases} 0 & n < 0 \\ \sum_{i=0}^{n} e^{-m}\frac{m^i}{i!} & n \geq 0 \end{cases}$$

Note: There are no *location* or *scale* parameters for the Poisson distribution.

Example

The following statement illustrates the CDF Poisson distribution function:

Statements	Results
`select cdf('POISSON', 2, 1);`	`0.919699`

See Also

Functions:

- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626
- "LOGSDF Function" on page 628
- "PDF POISSON Distribution Function" on page 728
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

CDF T Distribution Function

Returns a value from the T cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 774.

Syntax

CDF('T', *t*, *df* [, *nc*])

Arguments

t

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

df

is a numeric constant, variable, or expression that specifies the degrees of freedom.

Range *df* > 0

Data type DOUBLE

nc

is a numeric constant, variable, or expression that specifies an optional noncentrality parameter.

Data type DOUBLE

Details

The CDF function for the *T* distribution returns the probability that an observation from a *T* distribution, with degrees of freedom *df* and the noncentrality parameter *nc*, is less than or equal to *x*. This function accepts noninteger degrees of freedom. If *nc* is omitted or equal to zero, the value returned is from the central *T* distribution. In the following equation, let $\nu = df$ and let $\delta = nc$.

$$CDF('T', t, \nu, \delta) = \frac{1}{2^{(\nu/2-1)}\Gamma\left(\frac{\nu}{2}\right)} \int_0^{\infty} x^{\nu-1} e^{-\frac{1}{2}x^2} \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\frac{tx}{\sqrt{\nu}}} e^{-\frac{1}{2}(u-\delta)^2} du dx$$

Note: There are no *location* or *scale* parameters for the *T* distribution.

Example

The following statement illustrates the CDF T distribution function:

Statements	Results
`select cdf('T', .9, 5);`	`0.795314`

See Also

Functions:

- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “PDF T Distribution Function” on page 729
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

CDF TWEEDIE Distribution Function

Returns a value from the Tweedie cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 774.

Syntax

CDF ('TWEEDIE', *y*, *p* [, *μ*, *φ*])

Arguments

y

is a numeric constant, variable, or expression that specifies a random variable.

Range $y \geq 0$

Data type DOUBLE

Notes This argument is required.

When $p>1$, y is numeric. When $p=1$, y is a whole number.

p

is a numeric constant, variable, or expression that specifies the power parameter.

Range $p \geq 1$

Data type DOUBLE

Note This argument is required.

μ

is a numeric constant, variable, or expression that specifies the mean parameter.

Default 1

Range $\mu>0$

Data type DOUBLE

φ

is a numeric constant, variable, or expression that specifies the dispersion parameter.

Default 1

Range $\varphi>0$

Data type DOUBLE

Details

The CDF function for the Tweedie distribution returns an exponential dispersion model with variance and mean related by the equation *variance*= $\phi \times \mu^p$.

$$\int_0^y \frac{1}{y} \sum_{j=1}^{\infty} \left(\frac{y^{-j\alpha}(p-1)^{ja}}{\phi^{j(1-\alpha)}(2-p)^j j!\,\Gamma(-j\alpha)} \right) e\left(\frac{1}{\phi}\left(y\frac{\mu^{1-p}-1}{1-p} - \frac{\mu^{2-p}-1}{2-p}\right)\right) dy$$

The following relationship applies to the preceding equation:

$$\alpha = \frac{2-p}{1-p}$$

Note: The accuracy of computed Tweedie probabilities is highly dependent on the location in parameter space. Ten digits of accuracy are usually available except when *p* is near 2 or phi is near 0. In that case, the accuracy might be as low as six digits.

Example

The following statement illustrates the CDF Tweedie distribution function:

Statements	Results
`select cdf('TWEEDIE', .8, 5);`	`0.591763`

See Also

Functions:

- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “PDF TWEEDIE Distribution Function” on page 731

- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

CDF UNIFORM Distribution Function

Returns a value from the uniform cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 774.

Syntax

CDF('UNIFORM', *x* [, *l*, *r*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

l

is a numeric constant, variable, or expression that specifies the left location parameter.

Default 0

Data type DOUBLE

r

is a numeric constant, variable, or expression that specifies the right location parameter.

Default 1

Range *r* > *l*

Data type DOUBLE

Details

The CDF function for the uniform distribution returns the probability that an observation from a uniform distribution, with the left location parameter *l* and the right location parameter *r*, is less than or equal to *x*.

$$CDF('UNIFORM', x, l, r) = \begin{cases} 0 & x < l \\ \frac{x-l}{r-l} & l \le x < r \\ 1 & x \ge r \end{cases}$$

Note: The default values for *l* and *r* are 0 and 1, respectively.

Example

The following statement illustrates the CDF Uniform distribution function:

Statements	Results
`select cdf('UNIFORM', 0.25);`	`0.25`

See Also

Functions:

- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “PDF UNIFORM Distribution Function” on page 733
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

CDF WALD (Inverse Gaussian) Distribution Function

Returns a value from the Wald (also known as the inverse Gaussian) cumulative probability distribution.

Categories: CAS

Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 774.

Syntax

CDF('WALD', *x*, *λ* [, *μ*])
CDF('IGAUSS', *x*, *λ* [, *μ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $\lambda > 0$

Data type DOUBLE

μ

is a numeric constant, variable, or expression that specifies the mean parameter.

Default 1

Range $\mu > 0$

Data type DOUBLE

Details

The CDF function for the Wald distribution returns the probability that an observation from a Wald distribution, with the shape parameter *λ*, is less than or equal to *x*.

$$Fx(x) = \Phi\left\{\sqrt{\frac{\lambda}{x}}\left(\frac{x}{\mu} - 1\right)\right\} + e^{2\lambda/\mu}\Phi\left\{-\sqrt{\frac{\lambda}{x}}\left(\frac{x}{\mu} + 1\right)\right\}$$

In the equation, $\Phi(.)$ is the standard normal cumulative distribution function. When $x \le 0$, CDF is 0.

Example

The following statement illustrates the CDF Wald distribution function:

Statements	Results
`select cdf('WALD', 1, 2);`	`0.627698`

See Also

Functions:

- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “PDF Wald (Inverse Gaussian) Distribution Function” on page 735
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

CDF WEIBULL Distribution Function

Returns a value from the Weibull cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 774.

Syntax

CDF('WEIBULL', *x*, *a* [, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range *a* > 0

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The CDF function for the Weibull distribution returns the probability that an observation from a Weibull distribution, with the shape parameter *a* and the scale parameter λ, is less than or equal to *x*.

$$CDF('WEIBULL', x, a, \lambda) = \begin{cases} 0 & x < 0 \\ 1 - e^{-\left(\frac{x}{\lambda}\right)^a} & x \geq 0 \end{cases}$$

Example

The following statement illustrates the CDF Weibull distribution function:

Statements	Results
`select cdf('WEIBULL', 1, 2);`	`0.632121`

See Also

Functions:

- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626

- "LOGSDF Function" on page 628
- "PDF WEIBULL Distribution Function" on page 736
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

CEIL Function

Returns the smallest integer greater than or equal to a numeric value expression.

Categories:	CAS Truncation
Returned data type:	DECIMAL, DOUBLE, NUMERIC

Syntax

CEIL(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type	DECIMAL, DOUBLE, NUMERIC
See	"<sql-expression> Expression" on page 894 "FedSQL Expressions" on page 46

Details

If *expression* is null, then the CEILING function returns null. If the result is a number that does not fit into the range of the argument's data type, the CEIL function fails.

If the argument is DECIMAL, the result is DECIMAL. Otherwise, the argument is converted to DOUBLE (if not so already), and the result is DOUBLE.

Comparisons

Unlike the CEILZ function, the CEIL function fuzzes the result. If the argument is within 1E-12 of an integer, the CEIL function fuzzes the result to be equal to that

integer. The CEILZ function does not fuzz the result. Therefore, with the CEILZ function, you might get unexpected results.

Example

The following statements illustrate the CEIL function:

Statements	Results
`select ceil(-2.4);`	`-2`
`select ceil(1+1.e-11);`	`2`
`select ceil(-1+1.e-11);`	`0`
`select ceil(1+1.e-13);`	`1`

See Also

Functions:

- "CEILZ Function" on page 359
- "FLOOR Function" on page 520
- "FLOORZ Function" on page 521

CEILZ Function

Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing.

Categories: CAS
Truncation

Returned data type: DOUBLE

Syntax

CEILZ(*expression*)

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46.

Comparisons

Unlike the CEIL function, the CEILZ function uses zero fuzzing. If the argument is within 1E-12 of an integer, the CEIL function fuzzes the result to be equal to that integer. The CEILZ function does not fuzz the result. Therefore, with the CEILZ function, you might get unexpected results.

Example

The following statements illustrate the CEILZ function:

Statements	Results
`select ceilz(2.1);`	3
`select ceilz(3);`	3
`select ceilz(1+1.e-11);`	2
`select ceilz(223.456);`	224
`select ceilz(-223.456);`	-223

See Also

Functions:

- "CEIL Function" on page 358
- "FLOOR Function" on page 520
- "FLOORZ Function" on page 521

CHARACTER_LENGTH Function

Returns the number of characters in a string of any data type.

Categories:	Character CAS
Alias:	CHAR_LENGTH
Returned data type:	BIGINT
Note:	The CHARACTER_LENGTH function counts trailing blanks. If you do not want to count trailing blanks, then you must use the "TRIM Function" on page 844 to remove them.

Syntax

CHARACTER_LENGTH(*expression*)

Arguments

expression
specifies any valid expression.

Data type All data types are valid.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Comparisons

The CHARACTER_LENGTH function returns the number of characters in a character string, but counts a multibyte character as a single character. The OCTET_LENGTH function counts a multibyte character as multiple characters.

Example

The following statements illustrate the CHARACTER_LENGTH function:

Statements	Results
`select character_length('December');`	8

Statements	Results
`select character_length('FedSQL ');`	7

See Also

Functions:

- "OCTET_LENGTH Function" on page 689

CNONCT Function

Returns the noncentrality parameter from a chi-square distribution.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

CNONCT(*x*, *df*, *probability*)

Arguments

x
is a numeric random variable.

Range $x \geq 0$

Data type DOUBLE

df
is a numeric degrees of freedom parameter.

Range $df > 0$

Data type DOUBLE

probability
is a probability.

Range $0 < probability < 1$

Data type DOUBLE

Details

The CNONCT function returns the nonnegative noncentrality parameter from a noncentral chi-square distribution whose parameters are *x*, *df*, and *nc*. If *probability* is greater than the probability from the central chi-square distribution with the parameters *x* and *df*, a root to this problem does not exist. In this case a missing value is returned. A Newton-type algorithm is used to find a nonnegative root *nc* of the equation

$$P_c(x|\mathrm{df}, nc) - prob = 0$$

The following relationship applies to the preceding equation:

$$P_c(x|\mathrm{df}, \mathrm{nc}) = \varepsilon^{\frac{-nc}{2}} \sum_{j=0}^{\infty} \frac{\left(\frac{nc}{2}\right)^j}{j!} P_g\left(\frac{x}{2}\middle|\frac{df}{2} + j\right)$$

The following relationship applies to the preceding equation:

$$P_g(x|a)$$

is the probability from the gamma distribution given by

$$P_g(x|a) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} \varepsilon^{-t} dt$$

If the algorithm fails to converge to a fixed point, a missing value is returned.

Example

This example illustrates the CNONCT function.

```
proc ds2;
data test /overwrite=yes;
   dcl double x df nc prob;
   method init();
      x=2;
      df=4;
      do nc=1 to 3 by .5;
         prob=probchi(x, df, nc);
         output;
      end;
   end;
enddata;
run;
quit;

proc fedsql;
 select * from test;
 select cnonct(x, df, prob) from test;
quit;
```

Output 4.1 *Content of Table "Test"*

x	df	nc	prob
2	4	1	0.18611
2	4	1.5	0.155915
2	4	2	0.130477
2	4	2.5	0.109074
2	4	3	0.091092

Output 4.2 *Result of CNONCT Function*

CNONCT
1
1.5
2
2.5
3

See Also

Functions:

- "FNONCT Function" on page 525
- "TNONCT Function" on page 836

COALESCE Function

Returns the first non-null or nonmissing value from a list of numeric arguments.

Categories:	CAS Mathematical
Returned data type:	DOUBLE

Syntax

COALESCE(*expression*[, …*expression*])

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46.

Details

COALESCE accepts one or more numeric expressions. The COALESCE function checks the value of each expression in the order in which they are listed and returns the first non-null or nonmissing value. If only one value is listed, then the COALESCE function returns the value of that argument. If all the values of all expressions are null or missing, then the COALESCE function returns a null or a missing value depending on whether you are in ANSI mode or SAS mode. For more information, see “How FedSQL Processes Nulls and SAS Missing Values” on page 20.

Comparisons

The COALESCE function searches numeric expressions, whereas the COALESCEC function searches character expressions.

Example

The following statement illustrates the COALESCE function:

Statements	Results
`select COALESCE(., .A, 33, 22, 44, .);`	33

See Also

Functions:

- “COALESCEC Function” on page 366

COALESCEC Function

Returns the first non-null or nonmissing value from a list of character arguments.

Categories:	CAS Character
Returned data type:	CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

COALESCEC(*expression*[, …*expression*])

Arguments

expression

specifies any valid expression that evaluates or can be coerced to a character string.

Data type	CHAR, NCHAR, NVARCHAR, VARCHAR
See	“<sql-expression> Expression” on page 894 “FedSQL Expressions” on page 46.

Details

COALESCEC accepts one or more character expressions. The COALESCEC function checks the value of each expression in the order in which they are listed and returns the first non-null or nonmissing value. If only one value is listed, then the COALESCEC function returns the value of that expression. If all the values of all expressions are null or missing, then the COALESCEC function returns a null or missing value depending on whether you are in ANSI mode or SAS mode. For more information, see “How FedSQL Processes Nulls and SAS Missing Values” on page 20.

Comparisons

The COALESCEC function searches character expressions, whereas the COALESCE function searches numeric expressions.

Example

The following statements illustrate the COALESCEC function:

Statements	Results
`select coalescec('', 'Hello');`	`Hello`
`select coalescec('', 'Goodbye', 'Hello');`	`Goodbye`

See Also

Functions:

- "COALESCE Function" on page 364

COMB Function

Computes the number of combinations of *n* elements taken *r* at a time.

Categories: CAS
Combinatorial

Returned data type: DOUBLE

Syntax

COMB(*n*, *r*)

Arguments

n

is a nonnegative whole number that represents the total number of elements from which the sample is chosen.

Data type DOUBLE

r

is a nonnegative whole number that represents the number of chosen elements.

Restriction $r \leq n$

Data type DOUBLE

Details

The mathematical representation of the COMB function is given by the following equation:

$$COMB\left(n,r\right)=\binom{n}{r}=\frac{n!}{r!\left(n-r\right)!}$$

In the preceding equation, $n \geq 0$, $r \geq 0$, and $n \geq r$.

If the expression cannot be computed, a missing value is returned. For moderately large values, it is sometimes not possible to compute the COMB function.

Example

The following statement illustrates the COMB function:

Statement	Result
`select comb(27, 2);`	`351`

See Also

Functions:

- "FACT Function" on page 442

COMPARE Function

Returns the position of the leftmost character by which two strings differ, or returns 0 if there is no difference.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

COMPARE(*string-1*, *string-2*[, *modifiers*])

Arguments

string-1

specifies a character constant, variable, or expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

string-2

specifies a character constant, variable, or expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

modifiers

specifies a character string that can modify the action of the COMPARE function. You can use one or more of the following characters as a valid modifier:

i or I	ignores the case in *string-1* and *string-2*.
l or L	removes leading blanks in *string-1* and *string-2* before comparing the values.
n or N	removes quotation marks from any argument that is a name literal and ignores the case of *string-1* and *string-2*. A name literal is a name token that is expressed as a string within quotation marks, followed by the uppercase or lowercase letter *n*. Name literals enable you to use special characters (including blanks) that are not otherwise allowed in table or variable names. For COMPARE to recognize a string as a name literal, the first character must be a quotation mark.
:	(colon) truncates the longer of *string-1* or *string-2* to the length of the shorter string, or to one, whichever is greater. If you do not specify this modifier, the shorter string is padded with blanks to the same length as the longer string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip COMPARE ignores blanks that are used as modifiers.

Details

The Basics

The order in which the modifiers appear in the COMPARE function is relevant.

- “LN” first removes leading blanks from each string, and then removes quotation marks from name literals.
- “NL” first removes quotation marks from name literals, and then removes leading blanks from each string.

In the COMPARE function, if *string-1* and *string-2* do not differ, COMPARE returns a value of zero. If the arguments differ, then the following apply:

- The sign of the result is negative if *string-1* precedes *string-2* in a sort sequence, and positive if *string-1* follows *string-2* in a sort sequence.
- The magnitude of the result is equal to the position of the leftmost character at which the strings differ.

DBCS Compatibility

The DBCS equivalent function is KCOMPARE. There are minor differences between the COMPARE and KCOMPARE functions. Both functions accept varying numbers of arguments, but usage of the third argument is not compatible. The following example shows the differences in the syntax:

COMPARE(*string-1*, *string-2*[, *modifiers*])

KCOMPARE(*string-1*[, *position*[, *count*]], *string-2*)

For more information, see the “KCOMPARE Function” in *SAS National Language Support (NLS): Reference Guide*.

Examples

Example 1: Understanding the Order of Comparisons When Comparing Two Strings

The following program compares two strings by using the COMPARE function.

```
create table test(string1 CHAR(8) having informat $char8. format
$char8.,
                  string2 CHAR(8) having informat $char8. format
$char8.,
               modifiers CHAR(8) having informat $char8. format
$char8.);

  insert into test (string1, string2) values ('12345678', '12345678');
  insert into test (string1, string2) values ('123', 'abc');
  insert into test (string1, string2) values ('abc', 'abx');
  insert into test (string1, string2) values ('xyz', 'abcdef');
  insert into test (string1, string2) values ('aBc', 'abc');
```

```
  insert into test (string1, string2, modifiers) values ('aBc', 'AbC',
'i');
  insert into test (string1, string2, modifiers) values ('   abc  ',
'abc', '  ');
  insert into test (string1, string2, modifiers) values ('   abc  ',
'abc', '1');
  insert into test (string1, string2, modifiers) values (' abc    ',
'   abx  ', '  ');
  insert into test (string1, string2, modifiers) values(' abc    ',
'   abx  ', '1');

select string1, string2, modifiers, compare(string1, string2,
modifiers)
  from test;
```

Output 4.3 *Results of Comparing Two Strings By Using the COMPARE Function*

STRING1	STRING2	MODIFIERS	COMPARE
12345678	12345678		0
123	abc		-1
abc	abx		-3
xyz	abcdef		1
aBc	abc		-2
aBc	AbC	i	0
abc	abc		-1
abc	abc	1	-1
abc	abx		2
abc	abx	1	2

Example 2: Truncating Strings Using the COMPARE Function

The following program uses the : (colon) modifier to truncate strings.

```
create table test2 (pad1 double,
                    pad2 double,
                    truncate1 double,
                    truncate2 double,
                    blank double);

 insert into test2 (pad1, pad2, truncate1, truncate2, blank) values
(
     (select compare('abc','abc            ')),
     (select compare('abc','abcdef         ')),
```

```
        (select compare('abc','abcdef',':')),
        (select compare('abcdef','abc',':')),
        (select compare('','abc',          ':'))
    );

     select * from test2;
```

Output 4.4 *Results from Using the Colon Modifier to Truncate Strings*

PAD1	PAD2	TRUNCATE1	TRUNCATE2	BLANK
0	-4	0	0	-1

COMPBL Function

Removes multiple blanks from a character string.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

COMPBL(*character-expression*)

Arguments

character-expression
specifies any valid expression that evaluates or can be coerced to a character string and that specifies the character string to compress.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

Details

The COMPBL function removes multiple blanks in a character string by translating each occurrence of two or more consecutive blanks into a single blank.

Comparisons

The COMPRESS function removes every occurrence of the specific character from a string. If you specify a blank as the character to remove from the source string, the COMPRESS function is similar to the COMPBL function. However, the COMPRESS function removes all blanks from the source string. The COMPBL function compresses multiple blanks to a single blank and has no effect on a single blank.

Examples

Example 1: Removing Blanks from a String That Is Passed to the Function

The following program illustrates the COMPBL function:

```
data test (overwrite=yes);
   dcl char(18) string1 string2;
   method run();
      string1='125    E. Main St.';
      string2=compbl(string1);
      put string2;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
125 E. Main St.
```

Example 2

The following statements illustrate the COMPBL function.

Statements	Results
	`----+----1----+---2--`
`select compbl('January     Status');`	`January Status`
`select compbl('125    E. Main St.');`	`125 E. Main St.`

See Also

Functions:

■ "COMPRESS Function" on page 375

COMPOUND Function

Returns compound interest parameters.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

COMPOUND(*a*, *f*, *r*, *n*)

Arguments

a

specifies the initial amount.

Range $a \geq 0$

Data type DOUBLE

f

specifies the future amount (at the end of *n* periods).

Range $f \geq 0$

Data type DOUBLE

r

specifies the periodic interest rate expressed as a fraction.

Range $r \geq 0$

Data type DOUBLE

n

specifies the number of compounding periods.

Range $n \geq 0$

Data type DOUBLE

Details

The COMPOUND function returns the missing argument in the list of four arguments from a compound interest calculation. The arguments are related by the following equation:

$$f = a(1 + r)^n$$

One missing argument must be provided. A compound interest parameter is then calculated from the remaining three values. No adjustment is made to convert the results to round numbers.

If *n*=0, then

$$f = a$$

and

$$(1 + r)^n$$

is equal to 1.

Note: If you choose *r* as your missing value, then COMPOUND returns an error.

Example

The following statement illustrates the COMPOUND function. The accumulated value of an investment of $2000 at a nominal annual interest rate of 9%, compounded monthly after 30 months, can be expressed as follows:

```
select compound(2000,.,0.09/12,30);
```

The value returned is 2502.5435276.

COMPRESS Function

Returns a character string with specified characters removed from the original string.

Categories:	CAS Character
Returned data type:	CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

COMPRESS(*character-expression*[, *character-list-expression*] [, *modifier(s)*])

Arguments

character-expression

specifies any valid expression that evaluates to a character expression and from which specified characters will be removed.

Requirement Enclose a literal string of characters in single quotation marks.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

character-list-expression

specifies a variable or any valid expression that initializes a list of characters.

By default, the characters in this list are removed from *character-expression*.

Requirement Enclose a literal string of characters in single quotation marks.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

modifier

specifies a character constant, variable, or expression in which each non-blank character modifies the action of the COMPRESS function. Blanks are ignored. The following characters can be used as modifiers:

a or A	adds alphabetic characters to the list of characters.
c or C	adds control characters to the list of characters.
d or D	adds digits to the list of characters.
f or F	adds the underscore character and English letters to the list of characters.
g or G	adds graphic characters to the list of characters.
h or H	adds a horizontal tab to the list of characters.
i or I	ignores the case of the characters to be kept or removed.
k or K	keeps the characters in the list instead of removing them.
l or L	adds lowercase letters to the list of characters.
n or N	adds digits, the underscore character, and English letters to the list of characters.
o or O	processes the second and third arguments once rather than every time the COMPRESS function is called. Using the O modifier in the DATA step (excluding WHERE clauses), or in the SQL procedure, can make COMPRESS run much faster when you call it in a loop where the second and third arguments do not change.
p or P	adds punctuation marks to the list of characters.

s or S adds space characters (blank, horizontal tab, vertical tab, carriage return, line feed, form feed, and NBSP ('A0'x, or 160 decimal ASCII) to the list of characters.

t or T trims trailing blanks from the first and second arguments.

u or U adds uppercase letters to the list of characters.

w or W adds printable characters to the list of characters.

x or X adds hexadecimal characters to the list of characters.

Tip If the modifier is a constant, enclose it in single quotation marks. Specify multiple constants in a single set of quotation marks. Modifier can also be expressed as a variable or an expression.

Details

The COMPRESS function compiles a list of characters to keep or remove, comprising the characters in the second argument plus any types of characters that are specified by the modifiers.

Based on the number of arguments, the COMPRESS function works as follows:

Number of Arguments	Results
Only the first argument, *source*	All blanks have been removed. If the argument is completely blank, then the result is a string with a length of zero. If you assign the result to a character variable with a fixed length, then the value of that variable is padded with blanks to fill its defined length.
Two arguments, *source* and *chars*	All characters that appear in the second argument are removed from the result.
Three arguments, *source* , *chars*, and *modifier(s)*	The K modifier (specified in the third argument) determines whether the characters in the second argument are kept or removed from the result.

Here is an example that illustrates the effect of the third argument. The D modifier specifies digits. Both of the following function calls remove digits from the result:

```
compress(source, "1234567890");
compress(source, '', "d");
```

To remove digits and plus or minus signs, you could use the following function call:

```
COMPRESS(source, "1234567890+-");
compress(source, "+-", "d");
```

The COMPRESS function allows null values and SAS missing values in the second argument. A null or missing value is treated as a string that has a length of 0. A string that has length 0 is an empty string.

```
compress(source, null);
compress(source, '');
```

Example

These examples show how to remove blanks, characters, lowercase letters, and space characters from a string with the COMPRESS function. The last example shows how to use COMPRESS to keep characters from a string.

Statements	Results
	----+----1----+----2----+----
select compress('AB C D ');	ABCD
select compress('AB C D ', 'A ');	BCD
select compress('123-4567-8901 e 234-5678-9012 i', 'aeiou');	123-4567-8901 234-5678-9012
select compress('1 2 3 4 5', ' ','s');	12345
select compress('Math A English B Physics A', 'ABCD', 'A');	ABA

See Also

Functions:

- “COMPBL Function” on page 372
- “LEFT Function” on page 614
- “TRIM Function” in *SAS DS2 Language Reference*

CONSTANT Function

Computes machine and mathematical constants.

Categories:	CAS Mathematical
Returned data type:	DOUBLE

Syntax

CONSTANT(*constant*[, *parameter*])

Arguments

constant

is a character constant, variable, or expression that identifies the constant to be returned. Valid constants are as follows:

Constant	Description
`'E'`	The natural base
`'EULER'`	Euler constant
`'PI'`	Pi
`'EXACTINT' [,nbytes]`	Exact integer
`'BIG'`	The largest double-precision number
`'LOGBIG' [,base]`	The log with respect to *base* of BIG
`'SQRTBIG'`	The square root of BIG
`'SMALL'`	The smallest double-precision number
`'LOGSMALL' [,base]`	The log with respect to *base* of SMALL
`'SQRTSMALL'`	The square root of SMALL
`'MACEPS'`	Machine precision constant
`'LOGMACEPS' [,base]`	The log with respect to *base* of MACEPS
`'SQRTMACEPS'`	The square root of MACEPS

parameter

is a numeric parameter that can be used as an optional argument with some of the constants specified in *constant*. When used, *parameter* alters the functionality of the CONSTANT function.

Details

Overview

CAUTION
In some operating environments, the run-time library might have limitations that prevent the use of the full range of floating-point numbers that the hardware provides. In such cases, the CONSTANT function attempts to return values that are compatible with the limitations of the run-time library. For example, if the run-time library cannot compute `EXP(LOG(CONSTANT('BIG')))`, then `CONSTANT('LOGBIG')` will not return the same value as `LOG(CONSTANT('BIG'))`, but returns a value such that `EXP(CONSTANT('LOGBIG'))` can be computed.

The Natural Base

CONSTANT('E')

The natural base is described by the following equation:

$$\lim_{x \to 0} (1 + x)^{\frac{1}{x}} \approx 2.718281828459045$$

Euler Constant

CONSTANT('EULER')

Euler's constant is described by the following equation:

$$\lim_{n \to \infty} \left\{ \sum_{j=1}^{j=n} \frac{1}{j} - \log(n) \right\} \approx 0.577215664901532860$$

Pi

CONSTANT('PI')

Pi is the ratio between the circumference and the diameter of a circle. Many expressions exist for computing this constant. One such expression for the series is described by the following equation:

$$4 \sum_{j=0}^{j=\infty} \frac{(-1)^j}{2j+1} \approx 3.14159265358979323846$$

Exact Integer

CONSTANT('EXACTINT'[, *nbytes*])

Arguments

nbytes
is a numeric value that is the number of bytes.

Default 8

Range 2 ≤ *nbytes* ≤ 8

The exact integer is the largest integer *k* such that all integers less than or equal to *k* in absolute value have an exact representation in a SAS numeric variable of length *nbytes*. This information can be useful to know before you trim a SAS numeric variable from the default 8 bytes of storage to a lower number of bytes to save storage.

The Largest Double-Precision Number

CONSTANT('BIG')

This case returns the largest double-precision floating-point number (8-bytes) that is representable on your computer.

The Logarithm of BIG

CONSTANT('LOGBIG'[, *base*])

Arguments

base
is a numeric value that is the base of the logarithm.

Default the natural base, E

Restriction The *base* that you specify must be greater than the value of 1+SQRTMACEPS.

This case returns the logarithm with respect to *base* of the largest double-precision floating-point number (8-bytes) that is representable on your computer.

It is safe to exponentiate the given *base* raised to a power less than or equal to CONSTANT('LOGBIG', *base*) by using the power operation (**) without causing any overflows.

It is safe to exponentiate any floating-point number less than or equal to CONSTANT('LOGBIG') by using the exponential function, EXP, without causing any overflows.

The Square Root of BIG

CONSTANT('SQRTBIG')

This case returns the square root of the largest double-precision floating-point number (8-bytes) that is representable on your computer.

It is safe to square any floating-point number less than or equal to CONSTANT('SQRTBIG') without causing any overflows.

The Smallest Double-Precision Number

CONSTANT('SMALL')

This case returns the smallest double-precision floating-point number (8-bytes) that is representable on your computer.

The Logarithm of SMALL

CONSTANT('LOGSMALL'[, *base*])

Arguments

base
is a numeric value that is the base of the logarithm.

Default the natural base, E

Restriction The *base* that you specify must be greater than the value of 1+SQRTMACEPS.

This case returns the logarithm with respect to *base* of the smallest double-precision floating-point number (8-bytes) that is representable on your computer.

It is safe to exponentiate the given *base* raised to a power greater than or equal to CONSTANT('LOGSMALL', *base*) by using the power operation (**) without causing any underflows or 0.

It is safe to exponentiate any floating-point number greater than or equal to CONSTANT('LOGSMALL') by using the exponential function, EXP, without causing any underflows or 0.

The Square Root of SMALL

CONSTANT('SQRTSMALL')

This case returns the square root of the smallest double-precision floating-point number (8-bytes) that is representable on the computer.

It is safe to square any floating-point number greater than or equal to CONSTANT('SQRTBIG') without causing any underflows or 0.

Machine Precision

CONSTANT('MACEPS')

This case returns the smallest double-precision floating-point number (8-bytes) $\varepsilon = 2^{-j}$ for some integer *j*, such that $1 + \varepsilon > 1$.

This constant is important in finite precision computations.

The Logarithm of MACEPS

CONSTANT('LOGMACEPS'[, *base*])

Arguments

base
is a numeric value that is the base of the logarithm.

Default the natural base, E

Restriction The *base* that you specify must be greater than the value of 1+SQRTMACEPS.

This case returns the logarithm with respect to *base* of CONSTANT('MACEPS').

The Square Root of MACEPS

CONSTANT('SQRTMACEPS')

This case returns the square root of CONSTANT('MACEPS').

Example

The following example uses the CONSTANT function to return values for various constants.

```
 create table test(constant char(10), output double);
   insert into test(constant, output) values ('E', (select constant('E')));
   insert into test(constant, output) values ('EULER',(select constant('EULER')));
   insert into test(constant, output) values ('PI',(select constant('PI')));
   insert into test(constant, output) values ('EXACTINT', (select constant('EXACTINT')));
   insert into test(constant, output) values ('BIG', (select constant('BIG')));
   insert into test(constant, output) values ('LOGBIG',(select constant('LOGBIG')));
   insert into test(constant, output) values ('SQRTBIG', (select constant('SQRTBIG')));
   insert into test(constant, output) values ('SMALL', (select constant('SMALL')));
   insert into test(constant, output) values ('LOGSMALL', (select constant('LOGSMALL')));
   insert into test(constant, output) values ('SQRTSMALL', (select
constant('SQRTSMALL')));
   insert into test(constant, output) values ('MACEPS', (select constant('MACEPS')));
   insert into test(constant, output) values ('LOGMACEPS', (select
constant('LOGMACEPS')));
   insert into test(constant, output) values ('SQRTMACEPS',(select
constant('SQRTMACEPS')));

 select * from test;
```

Output 4.5 *Values Returned for Specified Constants*

CONSTANT	OUTPUT
E	2.718282
EULER	0.577216
PI	3.141593
EXACTINT	9.007E15
BIG	1.8E308
LOGBIG	709.7827
SQRTBIG	1.34E154
SMALL	223E-310
LOGSMALL	-708.396
SQRTSMALL	149E-156
MACEPS	2.22E-16
LOGMACEPS	-36.0437
SQRTMACEPS	1.49E-8

CONVX Function

Returns the convexity for an enumerated cash flow.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

CONVX(*y*, *f*, *c(1)*, …, *c(k)*)

Arguments

y

specifies the effective per-period yield-to-maturity.

Range $0 < y < 1$

Data type DOUBLE

Tip If you express *y* as a fraction, the dividend must be written as a decimal value. In DS2, integer division results in a value of zero. Zero is converted to a DOUBLE and is passed as the first argument to the CONVX function. The CONVX function returns missing when a zero is passed as the first parameter.

f

specifies the frequency of cash flows per period.

Range $f > 0$

Data type DOUBLE

c(1), ... , c(k)

specifies a list of cash flows.

Data type DOUBLE

Details

The CONVX function returns the value from the following equation.

$$C = \sum_{k=1}^{K} \frac{k(k+f)\dfrac{c(k)}{(1+y)^{\frac{k}{f}}}}{P\left((1+y)^2\right)f^2}$$

The following relationship applies to the preceding equation:

$$P = \sum_{k=1}^{K} \frac{c(k)}{(1+y)^{\frac{k}{f}}}$$

Example

The following statements illustrate the CONVX function:

Statements	Results
`select convx(1./20,.1,.33,.44, .55,.49,.50,.22,.4,.8,.01,.36,.2,.4);`	`976.9411`

See Also

Functions:

■ "CONVXP Function" on page 386

CONVXP Function

Returns the convexity for a periodic cash flow stream, such as a bond.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

CONVXP(*A*, *c*, *n*, *K*, k_0, *y*)

Arguments

A

specifies the par value.

Range $A > 0$

Data type DOUBLE

c

specifies the nominal per-period coupon rate, expressed as a decimal.

Range $0 \le c < 1$

Data type DOUBLE

n

specifies the number of coupons per period.

Range $n > 0$

Data type DOUBLE

K

specifies the number of remaining coupons.

Range $K > 0$

Data type DOUBLE

k_0

specifies the time from the present date to the first coupon date, expressed in terms of the number of periods.

Range $0 < k_0 \leq \frac{1}{n}$

Data type DOUBLE

y

specifies the nominal per-period yield-to-maturity.

Range $y > 0$

Data type DOUBLE

Details

The CONVXP function returns the value from the following equation.

$$C = \frac{1}{n^2}\left(\frac{\sum_{k=1}^{K} t_k(t_k+1)\frac{c(k)}{\left(1+\frac{y}{n}\right)^{t_k}}}{P\left(1+\frac{y}{n}\right)^2}\right)$$

The following relationships apply to the preceding equation:

$$t_k = nk_0 + k - 1$$

$$c(k) = \frac{c}{n}A \quad for\ \ k = 1, \ldots, K-1$$

$$c(K) = \left(1+\frac{c}{n}\right)A$$

The following relationship applies to the preceding equation:

$$P = \sum_{k=1}^{K} \frac{c(k)}{\left(1+\frac{y}{n}\right)^{t_k}}$$

Example

The following statements illustrate the CONVXP function:

Statements	Results
`select convxp(1000,.01,4,14,.33/2,.08);`	`11.729`

See Also

Functions:

- "CONVX Function" on page 384

COS Function

Returns the cosine in radians.

Categories:	Trigonometric CAS
Returned data type:	DOUBLE

Syntax

COS(*expression*)

Arguments

expression
specifies any valid SQL expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Example

The following statements illustrate the COS function:

Statements	Results
`select cos(0.5);`	`0.877583`
`select cos(0);`	`1`
`select cos(3.14159/3);`	`0.50003`

See Also

Functions:

- “ARCOS Function” on page 280
- “COSH Function” on page 389
- “SIN Function” on page 808
- “TAN Function” on page 829

COSH Function

Returns the hyperbolic cosine in radians.

Categories:	Trigonometric CAS
Returned data type:	DOUBLE

Syntax

COSH(*expression*)

Arguments

expression

specifies any valid SQL expression that evaluates to a numeric value.

Data type	DOUBLE
See	“<sql-expression> Expression” on page 894 “FedSQL Expressions” on page 46

Details

The COSH function returns the hyperbolic cosine of the argument, given by the following equation.

$$e^{argument} + e^{-argument}/2$$

Example

The following statements illustrate the COSH function:

Statements	Results
`select cosh(0);`	`1`
`select cosh(-5.0);`	`74.20995`
`select cosh(4.37);`	`39.52814`
`select cosh(0.5);`	`1.127626`

See Also

Functions:

- “ARCOS Function” on page 280
- “COS Function” on page 388
- “SINH Function” on page 809
- “TANH Function” on page 830

COT Function

Returns the tangent in radians.

Categories: Trigonometric
CAS

Returned data type: DOUBLE

Syntax

COT(*expression*)

Arguments

expression
specifies any valid SQL expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Example

The following statements illustrate the COT function:

Statements	Results
`select cot(0);`	.
`select cot(0.5)`	1.830488
`select cot(5.79);`	-1.86051
`select cot(1);`	0.642093

See Also

Functions:

- "ATAN Function" on page 287
- "COS Function" on page 388
- "TAN Function" on page 829

COUNT Function

Returns the number of rows retrieved by a SELECT statement for a specified table.

Categories:	Aggregate CAS
Alias:	N
Returned data type:	BIGINT

Syntax

Form 1: **COUNT**(*expression*)

Form 2: **COUNT**(*)

Form 3: **COUNT**([**DISTINCT**] *expression*)

Arguments

expression
specifies any valid SQL expression.

Data type All data types are valid.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

*****
returns a count of all rows from the table, including rows that contain null values or SAS missing values.

DISTINCT
(optional) eliminates duplicate rows from the aggregate calculation.

Details

You use the COUNT function in a SELECT statement to return the requested number of rows in a table.

The following list describes what is returned by using the different versions of the COUNT function:

Form 1: `COUNT(`*`expression`*`)`
returns the number of rows from a table that do not have a null value.

Form 2: `COUNT(*)`
returns the number of rows in a table.

Form 3: `COUNT(DISTINCT` *`expression`*`)`
returns the number of rows in *expression* that have unique values. SAS missing values are included in the results. Null values are not included in the results.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Example

Table: WORLDTEMPS on page 1184

The following statements illustrate the COUNT function:

Statements	Results
`select count(AvgHigh) from worldtemps;`	11
`select count(*) from worldtemps;`	12
`select count(distinct AvgHigh) from worldtemps;`	8

See Also

SELECT Statement Clauses:

- "SELECT Clause" on page 956
- "GROUP BY Clause" on page 968
- "HAVING Clause" on page 970

COUNTC Function

Counts the number of characters in a string that appear or do not appear in a list of characters.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

COUNTC(*string*, *charlist*[, *modifiers*])

Arguments

string

specifies a character constant, variable, or expression in which characters are counted.

Data type	CHAR, NCHAR, NVARCHAR, VARCHAR
Tip	Enclose a literal string of characters in quotation marks.

charlist

specifies a character constant, variable, or expression that initializes a list of characters. COUNTC counts characters in this list, provided that you do not specify the V modifier in the *modifiers* argument. If you specify the V modifier,

then all characters that are not in this list are counted. You can add more characters to the list by using other modifiers.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tips Enclose a literal string of characters in quotation marks.

If there are no characters in the list after processing the modifiers, COUNTC returns 0.

modifiers

specifies a character constant, variable, or expression in which each non-blank character modifies the action of the COUNTC function. Blanks are ignored. The following characters, in uppercase or lowercase, can be used as modifiers:

blank	is ignored.
a or A	adds alphabetic characters to the list of characters.
b or B	scans *string* from right to left, instead of from left to right.
c or C	adds control characters to the list of characters.
d or D	adds digits to the list of characters.
f or F	adds an underscore and English letters (that is, the characters that can begin a SAS variable name using VALIDVARNAME=V7) to the list of characters.
g or G	adds graphic characters to the list of characters.
h or H	adds a horizontal tab to the list of characters.
i or I	ignores case.
l or L	adds lowercase letters to the list of characters.
n or N	adds digits, an underscore, and English letters (that is, the characters that can appear in a SAS variable name using VALIDVARNAME=V7) to the list of characters.
o or O	processes the *charlist* and *modifier* arguments only once, at the first call to this instance of COUNTC. If you change the value of *charlist* or *modifier* in subsequent calls, the change might be ignored by COUNTC.
p or P	adds punctuation marks to the list of characters.
s or S	adds space characters to the list of characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed).
t or T	trims trailing blanks from *string* and *chars*. If you want to remove trailing blanks from only one character argument instead of both (or all) character arguments, use the TRIM function instead of the COUNTC function with the T modifier.
u or U	adds uppercase letters to the list of characters.
v or V	counts characters that do not appear in the list of characters. If you do not specify this modifier, then COUNTC counts characters that do appear in the list of characters.
w or W	adds printable characters to the list of characters.
x or X	adds hexadecimal characters to the list of characters.

Data type CHAR, VARCHAR

Tip If *modifier* is a constant, enclose it in quotation marks. Specify multiple constants in a single set of quotation marks.

Details

The COUNTC function allows character arguments to be null. Null arguments are treated as character strings with a length of zero. If there are no characters in the list of characters to be counted, COUNTC returns zero.

Note: Remember that strings with a CHAR data type are always padded out with blanks to the declared length. Strings with a VARCHAR data type return the length of the actual string instead of the declared length.

Comparisons

The COUNTC function counts individual characters in a character string, whereas the COUNT function counts substrings of characters in a character string.

Example

The following example uses the COUNTC function with and without modifiers to count the number of characters in a string.

```
create table test(string char(24));
 insert into test (string) values ('Baboons Eat Bananas');

 select string,
   countc(string, 'a') as a,
   countc(string, 'b') as b,
   countc(string, 'b', 'i') as b_i,
       countc(string,'abc','i') as abc_i,
   countc(string, 'abc', 'iv') as abc_iv,
   countc(string, 'abc', 'ivt') as abc_ivt
   from test;
```

Output 4.6 *Results from Using the COUNTC Function with and without Modifiers*

STRING	A	B	B_I	ABC_I	ABC_IV	ABC_IVT
Baboons Eat Bananas	5	1	3	8	16	11

See Also

Functions:

- "COUNTW Function" on page 396
- "INDEXC Function" on page 555
- "VERIFY Function" on page 855

COUNTW Function

Counts the number of words in a character string.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

COUNTW(*string*[, *chars*[, *modifiers*]])

Arguments

string

specifies a character constant, variable, or expression in which words are counted.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

chars

specifies an optional character constant, variable, or expression that initializes a list of characters. The characters in this list are the delimiters that separate words, provided that you do not use the K modifier in the *modifier* argument. If you specify the K modifier, then all characters that are not in this list are delimiters. You can add more characters to the list by using other modifiers.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

modifiers

specifies a character constant, variable, or expression in which each non-blank character modifies the action of the COUNTW function. The following characters, in uppercase or lowercase, can be used as modifiers:

blank	is ignored.
a or A	adds alphabetic characters to the list of characters.

b or B — counts from right to left instead of from left to right. Right-to-left counting makes a difference only when you use the Q modifier and the string contains unbalanced quotation marks.

c or C — adds control characters to the list of characters.

d or D — adds digits to the list of characters.

f or F — adds an underscore and English letters (that is, the characters that can begin a SAS variable name using VALIDVARNAME=V7) to the list of characters.

g or G — adds graphic characters to the list of characters.

h or H — adds a horizontal tab to the list of characters.

i or I — ignores the case of the characters.

k or K — causes all characters that are not in the list of characters to be treated as delimiters. If K is not specified, then all characters that are in the list of characters are treated as delimiters.

l or L — adds lowercase letters to the list of characters.

m or M — specifies that multiple consecutive delimiters, and delimiters at the beginning or end of the *string* argument, refer to words that have a length of zero. If the M modifier is not specified, then multiple consecutive delimiters are treated as one delimiter, and delimiters at the beginning or end of the *string* argument are ignored.

n or N — adds digits, an underscore, and English letters (that is, the characters that can appear after the first character in a SAS variable name using VALIDVARNAME=V7) to the list of characters.

o or O — processes the *chars* and *modifier* arguments only once, rather than every time the COUNTW function is called. Using the O modifier in the DATA step (excluding WHERE clauses), or in the SQL procedure, can make COUNTW run faster when you call it in a loop where *chars* and *modifier* arguments do not change.

p or P — adds punctuation marks to the list of characters.

q or Q — ignores delimiters that are inside substrings that are enclosed in quotation marks. If the value of *string* contains unmatched quotation marks, then scanning from left to right produces different words than scanning from right to left.

s or S — adds space characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed) to the list of characters.

t or T — trims trailing blanks from the *string* and *chars* arguments.

u or U — adds uppercase letters to the list of characters.

w or W — adds printable characters to the list of characters.

x or X — adds hexadecimal characters to the list of characters.

Data type CHAR, VARCHAR

Details

Definition of “Word”

In the COUNTW function, “word” refers to a substring that has one of the following characteristics:

- is bounded on the left by a delimiter or the beginning of the string
- is bounded on the right by a delimiter or the end of the string
- contains no delimiters, except if you use the Q modifier and the delimiters are within substrings that have quotation marks

Note: The definition of “word” is the same in both the SCAN function and the COUNTW function.

Delimiter refers to any of several characters that you can specify to separate words.

Using the COUNTW Function in ASCII and EBCDIC Environments

If you use the COUNTW function with only two arguments, the default delimiters depend on whether your computer uses ASCII or EBCDIC characters.

- If your computer uses ASCII characters, then the default delimiters are as follows:

  blank ! $ % & ( ) * + , - . / ; < ^ |

  In ASCII environments that do not contain the ^ character, the SCAN function uses the ~ character instead.
- If your computer uses EBCDIC characters, then the default delimiters are as follows:

  blank ! $ % & ( ) * + , - . / ; < ¬ | ¢

Using Null Arguments

The COUNTW function allows character arguments to be null. Null arguments are treated as character strings with a length of zero. Numeric arguments cannot be null.

Using the M Modifier

If you do not use the M modifier, then a word must contain at least one character. If you use the M modifier, then a word can have a length of zero. In this case, the number of words is one plus the number of delimiters in the string, not counting delimiters inside strings that are enclosed in quotation marks when you use the Q modifier.

Example

The following program shows how to use the COUNTW function with the M and P modifiers.

The explanation for the value of `mp` for each string is as follows:

- The period is the delimiter and the m modifier causes the period at the end to refer to a subsequent word with zero length, but still a word. So there is one word before the period and one word after the period for a total of two words.
- No delimiters, so there is only one word.
- The p modifier adds punctuation as a delimiter therefore 3 words.
- The p modifier adds punctuation, so / is a delimiter. The m modifier causes the leading / to refer to a word at beginning with zero length for a total of six words.
- The first \ is an escape character. The second \ is a delimiter, so there are six words.

```
proc fedsql;
drop table test;
create table test(string1 char(60) having informat $char60. format
$char60.);
describe table test;

 insert into test values ('The quick brown fox jumps over the lazy
dog.');
 insert into test values ('        Leading blanks');
 insert into test values ('2+2=4');
 insert into test values ('/unix/path/names/use/slashes');
 insert into test values ('\\Windows\\Path\\Names\\Use\\Backslashes');

select string1,
countw(string1) as basic,
countw(string1, ' ') as blank,
countw(string1, '.', 'mp') as mp from test;

quit;
```

Note: Double backslashes are used in the Windows pathname.

Output 4.7 *Results from Using the COUNTW Function with the M and P Modifiers*

STRING1	BASIC	BLANK	MP
The quick brown fox jumps over the lazy dog.	9	9	2
Leading blanks	2	2	1
2+2=4	2	1	3
/unix/path/names/use/slashes	5	1	6
\Windows\Path\Names\Use\Backslashes	1	1	6

See Also

Functions:

- "COUNTC Function" on page 393
- "SCAN Function" on page 795

CSC Function

Returns the cosecant.

Categories:	CAS Trigonometric
Returned data type:	DOUBLE

Syntax

CSC(*argument*)

Arguments

argument
specifies a numeric constant, variable, or expression and is expressed in radians.

Restriction *argument* cannot be 0 or a multiple of PI.

Data type DOUBLE

Comparisons

The CSC function is related to the SIN function in this way:

```
csc(x)=1/sin(x)
```

Example

The following statements illustrate the CSC function:

Statements	Results
`select csc(0.5);`	`2.08583`
`select csc(1);`	`1.188395`
`csc(3.14159/3);`	`1.154701`

Note: If you use `csc(0)`, then the CSC function returns a missing value. This is the correct behavior.

See Also

Functions:

- "COS Function" on page 388
- "COT Function" on page 390
- "SEC Function" on page 804
- "SIN Function" on page 808
- "TAN Function" on page 829

CSS Function

Returns the corrected sum of squares of all values in an expression.

Categories: Aggregate
Descriptive Statistics
CAS

Returned data type: DOUBLE

Syntax

CSS([DISTINCT] *expression*)

Arguments

expression
specifies any valid SQL expression.

Data type DOUBLE, FLOAT, REAL

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

DISTINCT
(optional) eliminates duplicate rows from the aggregate calculation.

Details

The corrected sum of squares is the sum of squared deviations (differences) from the mean.

Null values and SAS missing values are ignored and are not included in the computation.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Comparisons

The USS function returns the uncorrected sum of squares. The CSS function returns the corrected sum of squares of all values.

Example

Table: DENSITIES on page 1173

The following statement illustrates the CSS function:

Statements	Results
`select css(density) from densities;`	`211483.1`

See Also

Functions:

- "USS Function" on page 851

SELECT Statement Clauses:

- "SELECT Clause" on page 956
- "GROUP BY Clause" on page 968
- "HAVING Clause" on page 970

CUMIPMT Function

Returns the cumulative interest paid on a loan between the start and end period.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

CUMIPMT(*rate*, *number-of-periods*, *principal-amount*[, *start-period*][, *end-period*][, *type*])

Arguments

rate
specifies the interest rate per payment period.

Data type DOUBLE

number-of-periods
specifies the number of payment periods. *Number-of-periods* must be a positive, whole number.

Data type DOUBLE

principal-amount
specifies the principal amount of the loan. Zero is assumed if a missing value is specified.

Data type DOUBLE

start-period
specifies the start period for the calculation.

Data type DOUBLE

end-period
specifies the end period for the calculation.

Data type DOUBLE

type
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if *type* is omitted or if a missing value is specified.

Data type DOUBLE

Example

- The cumulative interest that is paid during the second year of a $125,000, 30-year loan with end-of-period monthly payments and a nominal annual interest rate of 9%, is computed as follows:

  ```
  select put(CUMIPMT(0.09/12, 360, 125000, 13, 24, 0), dollar10.2);
  ```

  This computation returns a value of $11,135.23.
- The interest that is paid on the first period of the same loan is computed in the following way:

  ```
  select put(CUMIPMT(0.09/12, 360, 125000, 1, 1, 0), dollar10.2);
  ```

  This computation returns a value of $937.50.

See Also

Functions:

- "CUMPRINC Function" on page 404

CUMPRINC Function

Returns the cumulative principal paid on a loan between the start and end period.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

CUMPRINC(*rate*, *number-of-periods*, *principal-amount*[, *start-period*] [, *end-period*][, *type*])

Arguments

rate
specifies the interest rate per payment period.

Data type DOUBLE

number-of-periods
specifies the number of payment periods.

Requirement *Number-of-periods* must be a positive whole number.

Data type DOUBLE

principal-amount
specifies the principal amount of the loan.

Data type DOUBLE

Note Zero is assumed if a missing or null value is specified.

start-period
specifies the start period for the calculation.

Data type DOUBLE

end-period
specifies the end period for the calculation.

Data type DOUBLE

type
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if *type* is omitted or if a missing value is specified.

Data type DOUBLE

Example

- The cumulative principal that is paid during the second year of a $125,000, 30-year loan with end-of-period monthly payments and a nominal annual interest rate of 9%, is computed as follows:

  ```
  select put(CUMPRINC(0.09/12, 360, 125000, 12, 24, 0), dollar10.2);
  ```

  This computation returns a value of $1008.23.

- The principal that is paid on the second year of the same loan with beginning-of-period payments is computed as follows:

  ```
  select put(CUMPRINC(0.09/12, 360, 125000, 12, 24, 1), dollar10.2);
  ```

  This computation returns a value of $1000.73.

See Also

Functions:

- "CUMIPMT Function" on page 403

CURRENT_DATE Function

Returns the current date for the time zone.

Categories:	Date and Time CAS
Returned data type:	DATE

Syntax

CURRENT_DATE

Comparisons

The CURRENT_DATE function returns the current date for the timezone. The CURRENT_TIMESTAMP_GMT() function returns the current GMT date.

Example

The following statement illustrates the CURRENT_DATE function:

Statements	Results
`select current_date;`	`05FEB2014`

See Also

Functions:

- "CURRENT_TIME Function" on page 407
- "CURRENT_TIME_GMT Function" on page 408

- “CURRENT_TIMESTAMP Function” on page 409
- “CURRENT_TIMESTAMP_GMT Function” on page 410

CURRENT_LOCALE Function

Returns the five character name of the current locale.

Categories: Character
CAS

Returned data type: CHAR(5)

Syntax

CURRENT_LOCALE()

Example

The following statement illustrates the CURRENT_LOCALE function:

Statements	Results
`select current_locale();`	`en_US`

CURRENT_TIME Function

Returns the current time for your time zone.

Categories: Date and Time
CAS

Alias: LOCALTIME

Returned data type: TIME

Syntax

CURRENT_TIME

Comparisons

The CURRENT_TIME function returns the current time for your time zone. The CURRENT_TIME_GMT function returns the current time in GMT.

Example

The following statement illustrates the CURRENT_TIME function:

Statements	Results
`select current_time;`	`11:00:49`
`select localtime;`	`11:00:49`

See Also

Functions:

- “CURRENT_DATE Function” on page 406
- “CURRENT_TIME_GMT Function” on page 408
- “CURRENT_TIMESTAMP Function” on page 409
- “CURRENT_TIMESTAMP_GMT Function” on page 410

CURRENT_TIME_GMT Function

Returns the current GMT time.

Categories:	Date and Time CAS
Returned data type:	TIME

Syntax

CURRENT_TIME_GMT()

Comparisons

The CURRENT_TIME_GMT function returns the current GMT time. The CURRENT_TIME function returns the current time for your time zone.

Example

The following statement illustrates the CURRENT_TIME_GMT function:

Statements	Results
`select current_time_gmt( );`	`15:00:49`

See Also

Functions:

- “CURRENT_DATE Function” on page 406
- “CURRENT_TIME Function” on page 407
- “CURRENT_TIMESTAMP Function” on page 409
- “CURRENT_TIMESTAMP_GMT Function” on page 410

CURRENT_TIMESTAMP Function

Returns the date and time for your time zone.

Categories:	Date and Time CAS
Alias:	LOCALTIMESTAMP
Returned data type:	TIMESTAMP

Syntax

CURRENT_TIMESTAMP

Comparisons

The CURRENT_TIMESTAMP function returns the date and time for your time zone.

Example

The following statement illustrates the CURRENT_TIMESTAMP function:

Statements	Results
`select current_timestamp;`	`05FEB2014:11:06:05`
`select localtimestamp;`	`05FEB2014:11:06:05`

See Also

Functions:

- “CURRENT_DATE Function” on page 406
- “CURRENT_TIME Function” on page 407
- “CURRENT_TIME_GMT Function” on page 408
- “CURRENT_TIMESTAMP_GMT Function” on page 410

CURRENT_TIMESTAMP_GMT Function

Returns the current GMT date and time.

Categories: Date and Time
CAS

Returned data type: TIMESTAMP

Syntax

CURRENT_TIMESTAMP_GMT()

Comparisons

The CURRENT_TIMESTAMP_GMT function returns the current GMT date and time. The CURRENT_TIMESTAMP function returns the current date and time for your time zone.

Example

The following statement illustrates the CURRENT_TIMESTAMP_GMT function:

Statements	Results
`select current_timestamp_gmt( );`	`05FEB2014:17:06:34`

See Also

Functions:

- “CURRENT_DATE Function” on page 406
- “CURRENT_TIME Function” on page 407
- “CURRENT_TIMESTAMP Function” on page 409
- “CURRENT_TIME_GMT Function” on page 408

CV Function

Returns the coefficient of variation.

Categories:	CAS Descriptive Statistics
Returned data type:	DOUBLE

Syntax

CV(*expression-1*, *expression-2* [, …*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement At least two arguments are required.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Example

The following statements illustrate the CV function:

Statements	Results
`select cv(5,9,3,6);`	`43.4782608695652`
`select cv(5,8,9,6,.);`	`26.0820265478651`
`select cv(8,9,6,.);`	`19.9242421519819`

DAIRY Function

Returns the derivative of the AIRY function.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

DAIRY(*x*)

Arguments

x
specifies a numeric constant, variable, or expression.

Details

The DAIRY function returns the value of the derivative of the AIRY function. (See a list of References.)

Example

The following program illustrates the DAIRY function:

Statements	Results
`select dairy(2.0);`	`-0.05309`
`select dairy(-2.0);`	`0.618259`

DATDIF Function

Returns the number of days between two dates after computing the difference between the dates according to specified day count conventions.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

DATDIF(*sdate*, *edate*, *basis*)

Arguments

sdate
specifies a SAS date value that identifies the starting date.

Data type DATE

Tip If *sdate* falls at the end of a month, then SAS treats the date as if it were the last day of a 30-day month.

edate
specifies a SAS date value that identifies the ending date.

Data type DATE

Tip If *edate* falls at the end of a month, then SAS treats the date as if it were the last day of a 30-day month.

basis

specifies a character string that represents the day count basis. The following values for *basis* are valid:

'30/360'

specifies a 30-day month and a 360-day year, regardless of the actual number of calendar days in a month or year.

A security that pays interest on the last day of a month will either always make its interest payments on the last day of the month, or it will always make its payments on the numerically same day of a month, unless that day is not a valid day of the month, such as February 30. For more information, see "Method of Calculation for Day Count Basis (30/360)" in *SAS Functions and CALL Routines: Reference*.

Alias '360'

'ACT/ACT'

uses the actual number of days between dates. Each month is considered to have the actual number of calendar days in that month, and each year is considered to have the actual number of calendar days in that year.

Alias *'Actual'*

'ACT/360'

uses the actual number of calendar days in a particular month, and 360 days as the number of days in a year, regardless of the actual number of days in a year.

Tip *ACT/360* is used for short-term securities.

'ACT/365'

uses the actual number of calendar days in a particular month, and 365 days as the number of days in a year, regardless of the actual number of days in a year.

Tip *ACT/365* is used for short-term securities.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The Basics

The DATDIF function has a specific meaning in the securities industry, and the method of calculation is not the same as the actual day count method. Calculations can use months and years that contain the actual number of days. Calculations can also be based on a 30-day month or a 360-day year. For more information about standard securities calculation methods, see the References section at the bottom of this function.

Note: When counting the number of days in a month, DATDIF *always* includes the starting date and excludes the ending date.

Method of Calculation for Day Count Basis (30/360)

To calculate the number of days between two dates, use the following formula:

$$Number of days = [(Y2 - Y1) * 360] + [(M2 - M1) * 30] + (D2 - D1)$$

Arguments

Y2
specifies the year of the later date.

Y1
specifies the year of the earlier date.

M2
specifies the month of the later date.

M1
specifies the month of the earlier date.

D2
specifies the day of the later date.

D1
specifies the day of the earlier date.

Because all months can contain only 30 days, you must adjust for the months that do not contain 30 days. Do this before you calculate the number of days between the two dates.

The following rules apply:

- If the security follows the End-of-Month rule, and D2 is the last day of February (28 days in a non-leap year, 29 days in a leap year), and D1 is the last day of February, then change D2 to 30.
- If the security follows the End-of-Month rule, and D1 is the last day of February, then change D1 to 30.
- If the value of D2 is 31 and the value of D1 is 30 or 31, then change D2 to 30.
- If the value of D1 is 31, then change D1 to 30.

Example

In the following example, DATDIF returns the actual number of days between two dates, as well as the number of days based on a 30-day month and a 360-day year.

This FedSQL SELECT statement returns the actual number of days between two dates:

```
select datdif(date'1978-10-16',date'1996-02-16', 'act/act');
```

This statement returns the number of days based on a 30-day month and a 360-day year.

```
select datdif(date'1978-10-16',date'1996-02-16','30/360');
```

The results are:

```
6332
6240
```

References

Securities Industry Association 1994. *Standard Securities Calculation Methods - Fixed Income Securities Formulas for Analytic Measures*. Vol. 2. New York, NY: Securities Industry Association.

DATE Function

Returns the current date as a SAS date value.

Categories:	CAS Date and Time
Alias:	TODAY
Returned data type:	DOUBLE

Syntax

DATE( )

Without Arguments

The DATE function has no arguments.

Comparisons

The DATE function does not take any arguments. The SAS date value returned is the number of days from January 1, 1960 to the current date.

For more information about how FedSQL handles dates, see “Dates and Times in FedSQL” on page 58.

Example

The following statement illustrates the DATE function:

Statements	Results
`select date();`	`19611`

See Also

Functions:

- "TODAY Function" on page 838

DATEJUL Function

Converts a Julian date to a SAS date value.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

DATEJUL(*julian-date*)

Arguments

julian-date

specifies any valid expression that evaluates to a numeric value and that represents a Julian date. A Julian date is a date in the form *yyddd* or *yyyyddd*, where *yy* or *yyyy* is a two-digit or four-digit whole number that represents the year and *ddd* is the number of the day of the year. The value of *ddd* must be between 1 and 365 (or 366 for a leap year).

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

A SAS date value is the number of days from January 1, 1960 to a specified date. The DATEJUL function returns the number of days from January 1, 1960 to the Julian date specified in *julian-date*.

For more information about how dates are handled in FedSQL, see "Dates and Times in FedSQL" on page 58.

Example

The following statements illustrate the DATEJUL function:

Statements	Results
`select datejul(11365);`	`18992`

See Also

Functions:

- "JULDATE Function" on page 606

DATEPART Function

Returns the date as year, month, and day.

Categories:	Date and Time CAS
Returned data type:	DATE

Syntax

DATEPART(*ts*)

Arguments

ts
specifies the timestamp.

Example

The following statement illustrates the DATEPART function:

Statements	Results
`select datepart(timestamp '2013-09-24 14:46:58');`	24SEP2013
`select put(datepart(d), date9.) from myoracle.test;`	24SEP2013
`select put(datepart(t), time12.) from myoracle.test;`	5:26:42

See Also

Functions:

- “MAKEDATE Function” on page 633
- “MAKETIME Function” on page 634
- “MAKETIMESTAMP Function” on page 635
- “TIMEPART Function” on page 831

DATETIME Function

Returns the current date and time of day as a SAS datetime value.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

DATETIME( )

Comparisons

The DATETIME function does not take any arguments. The SAS datetime value returned is the number of seconds from January 1, 1960 to the current date and time.

Example

The following statement illustrates the DATETIME function:

```
data test(overwrite=yes);
   dcl double dt ;
   method run();
      dt=datetime();
      put dt;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
1863786305.136
```

See Also

Concepts:

- "Dates and Times in FedSQL" on page 58

Functions:

- "DATE Function" on page 416

DAY Function

Returns the numeric day of the month from a date or datetime value.

Categories:	Date and Time CAS
Returned data type:	TINYINT

Syntax

DAY(*date* | *datetime*)

Arguments

date

specifies any valid expression that represents a date value.

Data type DATE

See "FedSQL Expressions" on page 46

datetime
specifies any valid expression that represents a datetime value.

Data type TIMESTAMP

See "FedSQL Expressions" on page 46

Example

Table: CUSTONLINE on page 1172

The following statement illustrates the DAY function:

Statements	Results
`select day(endtime) from custonline;`	1 2 15 1 1 2 16 1 1 15
`select day(current_time);`	17

See Also

- "Dates and Times in FedSQL" on page 58

Functions:

- "HOUR Function" on page 549
- "MINUTE Function" on page 649
- "MONTH Function" on page 655
- "SECOND Function" on page 806
- "YEAR Function" on page 864

DEGREES Function

Returns the number of degrees for an angle in radians.

Categories:	Trigonometric CAS
Returned data type:	DOUBLE

Syntax

DEGREES(*expression*)

Arguments

expression

specifies any valid expression that evaluates to an angle specified in radians.

Data type BIGINT, DOUBLE, FLOAT, INTEGER, REAL, SMALLINT, TINYINT.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

If *expression* is a null value, then the DEGREES function returns a null value. If the result is a number that does not fit into the range of a DOUBLE data type, the DEGREES function fails.

Example

The following statements illustrate the DEGREES function:

Statements	Results
`select degrees(2*pi());`	`360`
`select degrees(pi());`	`180`
`select degrees(pi()/2);`	`90`
`select degrees(pi()/4);`	`45`

See Also

Functions:

- "RADIANS Function" on page 779

DEQUOTE Function

Removes matching single quotation marks from a character string that begins with a single quotation mark, and deletes all characters to the right of the closing quotation mark.

Categories:	CAS Character
Returned data type:	CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

DEQUOTE(*expression*)

Arguments

expression

specifies any valid expression that evaluates or can be coerced to a character string.

Data type	CHAR, NCHAR, NVARCHAR, VARCHAR
See	"<sql-expression> Expression" on page 894 "FedSQL Expressions" on page 46

Details

The value that is returned by the DEQUOTE function depends on the first character or the first two characters in *expression*:

- If the first character of *expression* is not a quotation mark, DEQUOTE returns a syntax error.
- If the first character of *expression* is a single quotation mark, the DEQUOTE function removes that single quotation mark from the result. DEQUOTE then scans *expression* from left to right, looking for more single quotation marks or double quotation marks.

  All paired single quotation marks are replaced with a single quotation mark.

All paired double quotation marks are retained.

If a double quotation mark is the second character, DEQUOTE removes the double quotation mark from the result. DEQUOTE then scans *expression* from left to right. If a matching double quotation mark is found, the text between the double quotation marks is returned. Any text to the right of the closing double quotation mark, to the end of *expression* is removed from the result.

The first non-paired single quotation mark in *expression* is the closing single quotation mark and is removed.

If a close parentheses follows the close single quotation mark, the function returns the dequoted string. If characters exist to the right of the close single quotation mark, the function results in a syntax error and the error is printed in the SAS log.

- If *expression* is enclosed in double quotation marks, the DEQUOTE function returns a null or missing value.

Note: If *expression* is a constant enclosed in quotation marks, those quotation marks are not part of the value of *expression*. Therefore, you do not need to use DEQUOTE to remove the quotation marks that denote a constant.

Example

The following statements illustrate the DEQUOTE function:

Statements	Results
`select dequote(No quotation marks);`	`ERROR: Syntax error at or near "No"`
`select dequote(No 'leading' quotation marks);`	`ERROR: Syntax error at or near "No"`
`select dequote('Single matched quotation marks are removed');`	`Single matched quotation marks are removed`
`select dequote("Matched double quotation marks");`	`ERROR: Column "Matched double quotation marks" not found or cannot be accessed`
`select dequote('Paired ''single'' quotation marks are reduced to a single quotation mark');`	`Paired 'single' quotation marks are reduced to a single quotation mark`
`select dequote(' "Double quotation marks" within "single quotation marks", with space before open quotation mark');`	`"Double quotation marks" within "single quotation marks", with space before open quotation mark`
`select dequote('"Double quotation marks within single quotation marks, without space before open`	`Double quotation marks within single quotation marks, without space before open quotation mark`

Statements	Results
`quotation mark');`	
`select dequote('"Text after closing double quotation mark" is removed');`	`Text after closing double quotation mark`
`select dequote('No matching quotation mark);`	Statement execution does not complete. Submit the following characters to complete the execution: `');`
`select dequote('Identifiers after close quotation mark' results in a syntax error);`	`ERROR: Syntax error at or near "RESULTS"`

DEVIANCE Function

Returns the deviance based on a probability distribution.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

DEVIANCE(*distribution*, *variable*, *shape-parameter(s)*[, *ε*])

Arguments

distribution
is a character constant, variable, or expression that identifies the distribution. Valid distributions are listed in the following table:

Distribution	Argument
Bernoulli (p. 426)	`'BERNOULLI'` | `'BERN'`
Binomial (p. 426)	`'BINOMIAL'` | `'BINO'`
Gamma (p. 427)	`'GAMMA'`
Inverse Gauss (Wald) (p. 428)	`'IGAUSS'` | `'WALD'`

Distribution	Argument
Normal (p. 428)	'NORMAL' | 'GAUSSIAN'
Poisson (p. 428)	'POISSON' | 'POIS'

variable
is a numeric constant, variable, or expression.

shape-parameter(s)
are one or more distribution-specific numeric parameters that characterize the shape of the distribution.

ε
is an optional numeric small value used for all of the distributions, except for the normal distribution.

Details

The Bernoulli Distribution

DEVIANCE('BERNOULLI', *variable*, *p*[, *ε*])

Arguments

variable
is a binary numeric random variable that has the value of 1 for success and 0 for failure.

p
is a numeric probability of success with $\varepsilon \le p \le 1-\varepsilon$.

ε
is an optional positive numeric value that is used to bound *p*. Any value of *p* in the interval $0 \le p \le \varepsilon$ is replaced by *ε*. Any value of *p* in the interval $1 - \varepsilon \le p \le 1$ is replaced by $1 - \varepsilon$.

The DEVIANCE function returns the deviance from a Bernoulli distribution with a probability of success *p*, where success is defined as a random variable value of 1. The equation follows:

$$\text{DEVIANCE}('BERN', variable, p, \varepsilon) = \begin{cases} -2\log(1-p) & x = 0 \\ -2\log(p) & x = 1 \\ . & otherwise \end{cases}$$

The Binomial Distribution

DEVIANCE('BINO', *variable*, *μ*, *n*[, *ε*])

Arguments

variable
is a numeric random variable that contains the number of successes.

Range $0 \le variable \le 1$

μ
is a numeric mean parameter.

Range $n\varepsilon \le \mu \le n(1-\varepsilon)$

n
is a whole number of Bernoulli trials parameter

Range $n \ge 0$

ε
is an optional positive numeric value that is used to bound μ. Any value of μ in the interval $0 \le \mu \le n\varepsilon$ is replaced by $n\varepsilon$. Any value of μ in the interval $n(1 - \varepsilon) \le \mu \le n$ is replaced by $n(1 - \varepsilon)$.

The DEVIANCE function returns the deviance from a binomial distribution, with a probability of success p, and a number of independent Bernoulli trials n. The following equation describes the DEVIANCE function for the Binomial distribution, where x is the random variable:

$$\mathrm{DEVIANCE}('BINO', x, \mu, n) = \begin{cases} . & x < 0 \\ 2\left(x\log\left(\frac{x}{\mu}\right) + (n-x)\log\left(\frac{n-x}{n-\mu}\right)\right) & 0 \le x \le n \\ . & x > n \end{cases}$$

The Gamma Distribution

DEVIANCE('GAMMA', *variable*, *μ*[, *ε*])

Arguments

variable
is a numeric random variable.

Range $variable \ge \varepsilon$

μ
is a numeric mean parameter.

Range $\mu \ge \varepsilon$

ε
is an optional positive numeric value that is used to bound *variable* and μ. Any value of *variable* in the interval $0 \le variable \le \varepsilon$ is replaced by ε. Any value of μ in the interval $0 \le \mu \le \varepsilon$ is replaced by ε.

The DEVIANCE function returns the deviance from a gamma distribution with a mean parameter μ. The following equation describes the DEVIANCE function for the gamma distribution, where x is the random variable:

$$\mathrm{DEVIANCE}('GAMMA', x, \mu) = \begin{cases} . & x < 0 \\ 2\left(-\log\left(\frac{x}{\mu}\right) + \frac{x-\mu}{\mu}\right) & x \ge \varepsilon, \mu \ge \varepsilon \end{cases}$$

The Inverse Gauss (Wald) Distribution

DEVIANCE('IGAUSS' | 'WALD', *variable*, *μ*[, *ε*])

Arguments

variable
is a numeric random variable.

Range *variable* ≥ *ε*

μ
is a numeric mean parameter.

Range *μ* ≥*ε*

ε
is an optional positive numeric value that is used to bound *variable* and *μ*. Any value of *variable* in the interval 0 ≤ *variable* ≤ *ε* is replaced by *ε*. Any value of *μ* in the interval 0 ≤ *μ* ≤ *ε* is replaced by *ε*.

The DEVIANCE function returns the deviance from an inverse Gaussian distribution with a mean parameter *μ*. The following equation describes the DEVIANCE function for the inverse Gaussian distribution, where *x* is the random variable:

$$\text{DEVIANCE}(\text{'}IGAUSS\text{'}, x, \mu) = \begin{cases} . & x < 0 \\ \frac{(x-\mu)^2}{\mu^2 x} & x \geq \varepsilon, \mu \geq \varepsilon \end{cases}$$

The Normal Distribution

DEVIANCE('NORMAL' | 'GAUSSIAN', *variable*, *μ*)

Arguments

variable
is a numeric random variable.

μ
is a numeric mean parameter.

The DEVIANCE function returns the deviance from a normal distribution with a mean parameter *μ*. The following equation describes the DEVIANCE function for the normal distribution, where *x* is the random variable:

$$\text{DEVIANCE}(\text{'}NORMAL\text{'}, x, \mu) = (x-\mu)^2$$

The Poisson Distribution

DEVIANCE('POISSON', *variable*, *μ*[, *ε*])

Arguments

variable
is a numeric random variable.

Range *variable* ≥ 0

μ
is a numeric mean parameter.

Range $\mu \geq \varepsilon$

ε
is an optional positive numeric value that is used to bound μ. Any value of μ in the interval $0 \leq \mu \leq \varepsilon$ is replaced by ε.

The DEVIANCE function returns the deviance from a Poisson distribution with a mean parameter μ. The following equation describes the DEVIANCE function for the Poisson distribution, where x is the random variable:

$$\text{DEVIANCE}('POISSON', x, \mu) = \begin{cases} . & x < 0 \\ 2\left(x log\left(\frac{x}{\mu}\right) - (x - \mu)\right) & x \geq 0, \mu \geq \varepsilon \end{cases}$$

DHMS Function

Returns a SAS datetime value from date, hour, minute, and second values.

Categories:	CAS Date and Time
Returned data type:	DOUBLE

Syntax

DHMS(*date*, *hour*, *minute*, *second*)

Arguments

date
specifies any valid expression that represents a SAS date value.

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

hour
specifies a numeric expression that represents a whole number from 1 through 12.

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

minute
specifies a numeric expression that represents a whole number from 1 through 59.

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

second
specifies a numeric expression that represents a whole number from 1 through 59.

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

Details

The DHMS function returns a numeric value that represents a SAS datetime value. This numeric value can be either positive or negative.

Examples

Example 1: Using Datetime Values

The following program illustrates how to use the DHMS function with datetime values.

```
data test(overwrite=yes);
   dcl double dtid dtid1 dtid2 having format datetime.;
   method run();
      dtid=dhms(mdy(01, 03, 2018), 15, 30, 15);
      put dtid;
      dtid1=dhms(mdy(01, 03, 2018), 15, 30, 61);
      put dtid1;
      dtid2=dhms(mdy(01, 03, 2018), 15, .5, 15);
      put dtid2;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
03JAN18:15:30:15
03JAN18:15:31:01
03JAN18:15:00:45
```

Example 2

The following statements illustrate the DHMS function:

Statements	Results
`select dhms(mdy(08,10,2016),5,10,15);`	`1786425015`
`select put(dhms(date'2016-08-10', 5,10,15),datetime18.);`	`10AUG16:05:10:15`

See Also

Concepts:

- "Dates and Times in FedSQL" on page 58

Functions:

- "HMS Function" on page 544

DIGAMMA Function

Returns the value of the digamma function.

Categories:	CAS Mathematical
Returned data type:	DOUBLE

Syntax

DIGAMMA(*expression*)

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Restriction Zero and negative integers are not valid.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The DIGAMMA function returns the ratio that is given by the following equation.

$$\Psi(x) = \Gamma'(x)/\Gamma(x)$$

$\Gamma(.)$ and $\Gamma'(.)$ denote the gamma function and its derivative, respectively. For *expression*>0, the DIGAMMA function is the derivative of the lgamma function.

Example

The following statement illustrates the DIGAMMA function:

Statements	Results
`select digamma(1.0);`	`-0.577215665`

DUR Function

Returns the modified duration for an enumerated cash flow.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

DUR(*y*, *f*, *c(1), ..., c(k)*)

Arguments

y

specifies the effective per-period yield-to-maturity, expressed as a fraction.

Range $y > 0$

Data type DOUBLE

f

specifies the frequency of cash flows per period.

Range $f > 0$

Data type DOUBLE

c(1), ... ,c(k)

specifies a list of cash flows.

Data type DOUBLE

Details

The DUR function returns the value from the following equation.

$$C = \sum_{k=1}^{K} \frac{k\left(\frac{c(k)}{(1+y)^{\frac{k}{f}}}\right)}{(P(1+y)f)}$$

The following relationship applies to the preceding equation:

$$P = \sum_{k=1}^{K} \frac{c(k)}{(1+y)^{\frac{k}{f}}}$$

Example

The following statement illustrates the DUR function:

Statements	Results
`select dur(.05,1,.33,.44,.55, .49,.50,.22,.4,.8,.01,.36,.2,.4);`	`5.284025`

See Also

Functions:

- "DURP Function" on page 434

DURP Function

Returns the modified duration for a periodic cash flow stream, such as a bond.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

DURP(*A*, *c*, *n*, *K*, k_0, *y*)

Arguments

A

specifies the par value.

Range $A > 0$

Data type DOUBLE

c

specifies the nominal per-period coupon rate, expressed as a fraction.

Range $0 \le c < 1$

Data type DOUBLE

n

specifies the number of coupons per period.

Range $n > 0$ and is a whole number

Data type DOUBLE

K

specifies the number of remaining coupons.

Range $K > 0$ and is a whole number

Data type DOUBLE

k_0

specifies the time from the present date to the first coupon date, expressed in terms of the number of periods.

Range $0 < k_0 \le 1/n$

Data type DOUBLE

y

specifies the nominal per-period yield-to-maturity, expressed as a fraction.

Range $y > 0$

Data type DOUBLE

Details

The DURP function returns the value from the following equation.

$$D = \frac{1}{n} \frac{\sum_{k=1}^{K} t_k \frac{c(k)}{\left(1+\frac{y}{n}\right)^{t_k}}}{P\left(1+\frac{y}{n}\right)}$$

The following relationships apply to the preceding equation:

- $t_k = nk_0 + k - 1$
- $c(k) = \frac{c}{n}A \quad for\ k = 1, \ldots, K-1$
- $c(K) = \left(1+\frac{c}{n}\right)A$

The following relationship applies to the preceding equation:

$$P = \sum_{k=1}^{K} \frac{c(k)}{\left(1+\frac{y}{n}\right)^{t_k}}$$

Example

The following statement illustrates the DURP function:

Statements	Results
`select durp(1000,1/100,4,14,.33/2,.10);`	`3.331707`

See Also

Functions:

- “DUR Function” on page 432

E Function

Returns the natural logarithm, e.

Categories:	Mathematical CAS
Returned data type:	DOUBLE

Syntax

E()

Details

The E function returns the value of the natural logarithm, e, 2.7182818.

Comparisons

The E function takes no argument and returns the value 2.7182818. The EXP function takes an argument and raises e to the power that is supplied by the argument.

Example

The following statement illustrates the E function:

Statements	Results
`select e();`	`2.718282`

See Also

Functions:

- “EXP Function” on page 441

- "LOG Function" on page 617

EFFRATE Function

Returns the effective annual interest rate.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

EFFRATE(*compounding-interval*, *rate*)

Arguments

compounding-interval
is a SAS interval. This value represents how often *rate* compounds.

Data type CHAR

rate
is numeric. *Rate* is a nominal annual interest rate (expressed as a percentage) that is compounded at each compounding interval.

Data type DOUBLE

Details

The EFFRATE function returns the effective annual interest rate. The function computes the effective annual interest rate that corresponds to a nominal annual interest rate.

The following details apply to the EFFRATE function:

- The values for rates must be at least –99.
- In considering a nominal interest rate and a compounding interval, if *compounding-interval* is 'CONTINUOUS', then the value that is returned by EFFRATE equals $e^{rate/100}-1$.

  If *compounding-interval* is not 'CONTINUOUS', and *m* compounding intervals occur in a year, the value that is returned by EFFRATE equals $(1+[rate/100\ m])^{m-1}$.
- The following values are valid for *compounding-interval*:
  - 'CONTINUOUS'

- 'DAY'
- 'SEMIMONTH'
- 'MONTH'
- 'QUARTER'
- 'SEMIYEAR'
- 'YEAR'

- If the interval is 'DAY', then *m*=365.

Example

The following programs show how the effective rate is calculated:

- If a nominal rate is 10%, this program shows the corresponding effective rate when interest is compounded monthly.

  ```
  select EFFRATE('MONTH', 10);
  ```

  The effective rate is 10.4713067441296.

- If a nominal rate is 10%, this program shows the corresponding effective rate when interest is compounded quarterly.

  ```
  select EFFRATE('QUARTER', 10);
  ```

  The effective rate is 10.3812890624999.

ERF Function

Returns the value of the (normal) error function.

Categories:	CAS Mathematical
Returned data type:	DOUBLE

Syntax

ERF(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The ERF function returns the integral, given by the following:

$$\mathrm{ERF}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \varepsilon^{-z^2} dz$$

You can use the ERF function to find the probability (p) that a normally distributed random variable with mean 0 and standard deviation takes on a value less than X. For example, the quantity that is given by the following statement is equivalent to PROBNORM(X):

```
p=.5+.5*erf(x/sqrt(2));
```

Example

The following statements illustrate the ERF function:

Statement	Result
`select erf(1.0);`	`0.8427007929`
`select erf(-1.0);`	`-0.842700793`

See Also

Functions:

- "ERFC Function" on page 439
- "PROBNORM Function" on page 767

ERFC Function

Returns the value of the complementary (normal) error function.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

ERFC(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The ERFC function returns the complement to the ERF function (that is, 1 − ERF(*argument*)).

Example

The following statements illustrate the ERFC function:

Statement	Result
`select erfc(1.0);`	`0.1572992071`
`select erfc(-1.0);`	`1.8427007929`

See Also

Functions:

- "ERF Function" on page 438

EXP Function

Returns the value of the e constant raised to a specified power.

Categories:	Mathematical CAS
Returned data type:	DOUBLE

Syntax

EXP(*expression*)

Arguments

expression

specifies any valid SQL expression that evaluates to a numeric value.

Data type	BIGINT, DOUBLE, FLOAT, INTEGER, REAL, SMALLINT, TINTYINT
See	"<sql-expression> Expression" on page 894 "FedSQL Expressions" on page 46

Details

The EXP function raises the constant *e*, which is approximately given by 2.71828, to the power that is supplied by the argument. The result is limited by the maximum value of a double decimal value on the computer.

Comparisons

The EXP function takes an argument and raises *e* to the power that is supplied by the argument. The E function takes no argument and returns the value 2.7182818.

Example

The following statements illustrate the EXP function:

Statements	Results
`select exp(1.0);`	`2.718282`
`select exp(0);`	`1`

See Also

Functions:

- "E Function" on page 436
- "LOG Function" on page 617

FACT Function

Computes a factorial.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

FACT(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The mathematical representation of the FACT function is given by the following equation:

$$FACT(n) = n!$$

In this equation, $n \geq 0$.

If the expression cannot be computed, a missing value is returned. For moderately large values, it is sometimes not possible to compute the FACT function.

Example

The following statement illustrates the FACT function:

Statement	Result
`select fact(5);`	`120`

See Also

Functions:

- "COMB Function" on page 367

FINANCE Function

Computes financial calculations such as depreciation, maturation, accrued interest, net present value, periodic savings, and internal rates of return.

Categories: CAS
Financial

Syntax

FINANCE(*string-identifier*, *parameter–1*, *parameter–2*, …)

Arguments

string-identifier
specifies a character constant, variable, or expression. Valid values for *string-identifier* are listed in the following table.

string-identifier	Description
'ACCRINT'	computes the accrued interest for a security that pays periodic interest.
'ACCRINTM'	computes the accrued interest for a security that pays interest at maturity.
' AMORDEGRC'	computes the depreciation for each accounting period by using a depreciation coefficient.
'AMORLINC'	computes the depreciation for each accounting period.
'COUPDAYBS'	computes the number of days from the beginning of the coupon period to the settlement date.
'COUPDAYS'	computes the number of days in the coupon period that contains the settlement date.
'COUPDAYSNC'	computes the number of days from the settlement date to the next coupon date.
'COUPNCD'	computes the next coupon date after the settlement date.
'COUPNUM'	computes the number of coupons that are payable between the settlement date and the maturity date.
'COUPPCD'	computes the previous coupon date before the settlement date.
'CUMIPMT'	computes the cumulative interest that is paid between two periods.
'CUMPRINC'	computes the cumulative principal that is paid on a loan between two periods.
'DB'	computes the depreciation of an asset for a specified period by using the fixed-declining balance method.
'DDB'	computes the depreciation of an asset for a specified period by using the double-declining balance method or some other method that you specify.
'DISC'	computes the discount rate for a security.
'DOLLARDE'	converts a dollar price, expressed as a fraction, to a dollar price, expressed as a decimal number.
'DOLLARFR'	converts a dollar price, expressed as a decimal number, to a dollar price, expressed as a fraction.

string-identifier	Description
'DURATION'	computes the annual duration of a security with periodic interest payments.
'EFFECT'	computes the effective annual interest rate.
'FV'	computes the future value of an investment.
'FVSCHEDULE'	computes the future value of an initial principal after applying a series of compound interest rates.
'INTRATE'	computes the interest rate for a fully invested security.
'IPMT'	computes the interest payment for an investment for a given period.
'IRR'	computes the internal rate of return for a series of cash flows.
'ISPMT'	calculates the interest paid during a specific period of an investment.
'MDURATION'	computes the Macaulay modified duration for a security with an assumed face value of $100.
'MIRR'	computes the internal rate of return where positive and negative cash flows are financed at different rates.
'NOMINAL'	computes the annual nominal interest rate.
'NPER'	computes the number of periods for an investment.
'NPV'	computes the net present value of an investment based on a series of periodic cash flows and a discount rate.
'ODDFPRICE'	computes the price per $100 face value of a security with an odd first period.
'ODDFYIELD'	computes the yield of a security with an odd first period.
'ODDLPRICE'	computes the price per $100 face value of a security with an odd last period.
'ODDLYIELD'	computes the yield of a security with an odd last period.
'PMT'	computes the periodic payment for an annuity.

string-identifier	Description
'PPMT'	computes the payment on the principal for an investment for a given period.
'PRICE'	computes the price per $100 face value of a security that pays periodic interest.
'PRICEDISC'	computes the price per $100 face value of a discounted security.
'PRICEMAT'	computes the price per $100 face value of a security that pays interest at maturity.
'PV'	computes the present value of an investment.
'RATE'	computes the interest rate per period of an annuity.
'RECEIVED'	computes the amount received at maturity for a fully invested security.
'SLN'	computes the straight-line depreciation of an asset for one period.
'SYD'	computes the sum-of-years digits depreciation of an asset for a specified period.
'TBILLEQ'	computes the bond-equivalent yield for a treasury bill.
'TBILLPRICE'	computes the price per $100 face value for a treasury bill.
'TBILLYIELD'	computes the yield for a treasury bill.
'VDB'	computes the depreciation of an asset for a specified or partial period by using a declining balance method.
'XIRR'	computes the internal rate of return for a schedule of cash flows that is not necessarily periodic.
'XNPV'	computes the net present value for a schedule of cash flows that is not necessarily periodic.
'YIELD'	computes the yield on a security that pays periodic interest.
'YIELDDISC'	computes the annual yield for a discounted security (for example, a treasury bill).
'YIELDMAT'	computes the annual yield of a security that pays interest at maturity.

parameter
specifies a parameter that is associated with each *string-identifier*. The following parameters are available:

basis
is an optional parameter that specifies a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

interest-rates
specifies rates that are provided as numeric values and not as percentages.

dates
specifies that all dates in the financial functions are SAS dates.

sign-of-cash-values
for all the arguments, specifies that the cash that you pay out, such as deposits to savings or other withdrawals, is represented by negative numbers. It also specifies that the cash that you receive, such as dividend checks and other deposits, is represented by positive numbers.

FINANCE ACCRINT Function

Computes the accrued interest for a security that pays periodic interest.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('ACCRINT', *issue*, *first-interest*, *settlement*, *rate*, *par-value*, *frequency*, [*basis*]);

Arguments

issue
specifies the issue date of the security.

Requirement *Issue* is a SAS date.

Data type DOUBLE

first-interest
specifies the first interest date of the security.

Requirement *First-interest* is a SAS date.

Data type DOUBLE

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

par-value
specifies the par value of the security. If you omit *par-value*, SAS uses the value `$1000`.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365

Numeric Value	String Value	Day Count Method
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Accrued Interest: ACCRINT

The following example computes the accrued interest for a security that pays periodic interest.

Statements	Results
`select finance('accrint', mdy(2, 28, 2016), mdy(8, 31, 2016), MDY(5, 1, 2016), 0.1, 1000, 2, 1);`	`17.12255`

FINANCE ACCRINTM Function

Computes the accrued interest for a security that pays interest at maturity.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('ACCRINTM', *issue*, *settlement*, *rate*, *par-value*, [*basis*]);

Arguments

issue
specifies the issue date of the security.

Requirement *Issue* is a SAS date.

Data type DOUBLE

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

par-value
specifies the par value of the security. If you omit *par-value*, SAS uses the value `$1000`.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Accrued Interest: ACCRINTM

The following example computes the accrued interest for a security that pays interest at maturity.

Statements	Results
`select finance('accrintm', mdy(2, 28, 2015), mdy(8, 31, 2015), 0.1, 1000, 0);`	`50.55556`

FINANCE AMORDEGRC Function

Computes the depreciation for each accounting period by using a depreciation coefficient.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

FINANCE('AMORDEGRC', *cost*, *date-purchased*, *first-period*, *salvage*, *period*, *rate*, [*basis*]);

Arguments

cost
specifies the initial cost of the asset.

Data type DOUBLE

date-purchased
specifies the date of the purchase of the asset.

Requirement *Date-purchased* is a SAS date.

Data type DOUBLE

first-period
specifies the date of the end of the first period.

Requirement *First-period* is a SAS date.

Data type DOUBLE

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

Data type DOUBLE

period
specifies the depreciation period.

Data type DOUBLE

rate
specifies the rate of depreciation.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

TIP When the first argument of the FINANCE function is AMORDEGRC and the value of *basis* is 2, the function returns a missing value.

Data type DOUBLE

Example: Computing Depreciation: AMORDEGRC

The following example computes the depreciation for each accounting period by using a depreciation coefficient.

Statements	Results
`select finance('amordegrc', 2400, mdy(8, 19, 2016), mdy(12, 31, 2016), 300, 1, 0.15, 1);`	`776`

FINANCE AMORLINC Function

Computes the depreciation for each accounting period.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('AMORLINC', *cost*, *date-purchased*, *first-period*, *salvage*, *period*, *rate*, [*basis*]);

Arguments

cost
specifies the initial cost of the asset.

Data type DOUBLE

date-purchased
specifies the date of the purchase of the asset.

Requirement *Date-purchased* is a SAS date.

Data type DOUBLE

first-period
specifies the date of the end of the first period.

Requirement *First-period* is a SAS date.

Data type DOUBLE

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

Data type DOUBLE

period
specifies the depreciation period.

Data type DOUBLE

rate
specifies the rate of depreciation.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360

Numeric Value	String Value	Day Count Method
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

TIP When the first argument of the FINANCE function is AMORLINC and the value of *basis* is 2, the function returns a missing value.

Data type DOUBLE

Example: Computing Description: AMORLINC

The following example computes the depreciation for each accounting period.

Statements	Results
`select finance('AMORLINC', 2400, mdy(9, 30, 2016), mdy(12, 31, 2016), 245, 0, 0.115, 0);`	`69`

FINANCE COUPDAYBS Function

Computes the number of days from the beginning of the coupon period to the settlement date.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('COUPDAYBS', *settlement*, *maturity*, *frequency*, [*basis*]);

Arguments

settlement

specifies the settlement date of the security. The security settlement date is the date after the issue date when the security is traded to the buyer.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity

specifies the maturity date of the security. The maturity date is the date on which the security expires.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

frequency

specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis

specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: COUPDAYBS

The following example computes the number of days from the beginning of the coupon period to the settlement date.

Statements	Results
`select finance('COUPDAYBS', mdy(12,30,2013), mdy(11,29,2016), 4, 2);`	31

FINANCE COUPDAYS Function

Computes the number of days in the coupon period that contains the settlement date.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('COUPDAYS', *settlement*, *maturity*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: COUPDAYS

The following example computes the number of days in the coupon period that contains the settlement date.

Statements	Results
`select finance('COUPDAYS', mdy(1,25,2015), mdy(11,15,2016), 2, 1);`	`181`

FINANCE COUPDAYSNC Function

Computes the number of days from the settlement date to the next coupon date.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('COUPDAYSNC', *settlement*, *maturity*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: COUPDAYSNC

The following example computes the number of days from the settlement date to the next coupon date.

Statements	Results
`select FINANCE('COUPDAYSNC', mdy(1,25,2007),`	110

Statements	Results
`mdy(11,15,2008), 2, 1);`	

FINANCE COUPNCD Function

Computes the next coupon date after the settlement date.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('COUPNCD', *settlement*, *maturity*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360

Numeric Value	String Value	Day Count Method
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: COUPNCD

The following example computes the next coupon date after the settlement date.

Statements	Results
`select FINANCE('COUPNCD', mdy(1, 25, 2007), mdy(11, 15, 2008), 2, 1);`	`17301`
`select put(FINANCE('COUPNCD', mdy(1, 25, 2007), mdy(11, 15, 2008), 2, 1), date7.);`	`15MAY07`

FINANCE COUPNUM Function

Computes the number of coupons that are payable between the settlement date and the maturity date.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('COUPNUM', *settlement*, *maturity*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: COUPNUM

The following example computes the number of coupons that are payable between the settlement date and the maturity date.

Statements	Results
`select FINANCE('COUPNUM', mdy(1, 25, 2015),`	4

Statements	Results
`mdy(11,15,2016), 2, 1);`	

FINANCE COUPPCD Function

Computes the previous coupon date before the settlement date.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('COUPPCD', *settlement*, *maturity*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360

Numeric Value	String Value	Day Count Method
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: COUPPCD

The following example computes the previous coupon date before the settlement date.

Statements	Results
`select FINANCE('COUPPCD', mdy(1, 25, 2007), mdy(11, 15, 2008), 2, 1);`	`1720`
`select put(FINANCE('COUPPCD', mdy(1, 25, 2007), mdy(11, 15, 2008), 2, 1), date7.);`	`15NOV06`

FINANCE CUMIPMT Function

Computes the cumulative interest paid between two periods.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('CUMIPMT', *rate*, *nper*, *pv*, *start-period*, *end-period*, [*type*]);

Arguments

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

nper
specifies the total number of payment periods.

Data type DOUBLE

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently.

Data type DOUBLE

start-period
specifies the first period in the calculation. Payment periods are numbered beginning with 1.

Data type DOUBLE

end-period
specifies the last period in the calculation.

Data type DOUBLE

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

If payments are due at the end of the period, then either omit the *type* argument or set it to 0. If payments are due at the beginning of the period, then set *type* to 1.

Data type DOUBLE

Example: Computing Description: CUMIPMT

The following example computes the cumulative interest that is paid between two periods.

Statements	Results
`select FINANCE('CUMIPMT', 0.09, 30, 125000, 13, 24, 0);`	`-94054.8`

FINANCE CUMPRINC Function

Computes the cumulative principal that is paid on a loan between two periods.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

FINANCE('CUMPRINC', *rate*, *nper*, *pv*, *start-period*, *end-period*, [*type*]);

Arguments

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

nper
specifies the total number of payment periods.

Data type DOUBLE

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently.

Data type DOUBLE

start-period
specifies the first period in the calculation. Payment periods are numbered beginning with 1.

Data type DOUBLE

end-period
specifies the last period in the calculation.

Data type DOUBLE

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

If payments are due at the end of the period, then either omit the *type* argument or set it to 0. If payments are due at the beginning of the period, then set *type* to 1.

Data type DOUBLE

Example: Computing Description: CUMPRINC

The following example computes the cumulative principal that is paid on a loan between two periods.

Statements	Results
`select FINANCE('CUMPRINC', 0.09, 30, 125000, 13, 24, 0);`	`-51949.7`

FINANCE DB Function

Computes the depreciation of an asset for a specified period by using the fixed-declining balance method.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('DB', *cost*, *salvage*, *life*, *period*, [*month*]);

Arguments

cost
specifies the initial cost of the asset.

Data type DOUBLE

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

Data type DOUBLE

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

Data type DOUBLE

period
specifies the period for which you want to calculate the depreciation. *Period* must use the same time units as *life*.

Data type DOUBLE

month
specifies the number of months (month is an optional numeric argument). If month is omitted, it defaults to a value of 12.

Data type DOUBLE

Example: Computing Description: DB

The following example computes the depreciation of an asset for a specified period by using the fixed-declining balance method.

Statements	Results
`select FINANCE('DB', 1000000, 100000, 6, 2, 7);`	`259639.4`

FINANCE DDB Function

Computes the depreciation of an asset for a specified period by using the double-declining balance method or some other method that you specify.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('DDB', *cost*, *salvage*, *life*, *period*, [*factor*]);

Arguments

cost
specifies the initial cost of the asset.

Data type DOUBLE

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

Data type DOUBLE

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

Data type DOUBLE

period
specifies the period for which you want to calculate the depreciation. *Period* must use the same time units as *life*.

Data type DOUBLE

factor
specifies the rate at which the balance declines. If *factor* is omitted, it is assumed to be 2 (the double-declining balance method).

Data type DOUBLE

Example: Computing Description: DDB

The following example computes the depreciation of an asset for a specified period by using the double-declining balance method or some other method that you specify.

Statements	Results
`select FINANCE('DDB', 2400, 300, 10*365, 1, .)`	`1.315068`

FINANCE DISC Function

Computes the discount rate for a security.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('DISC', *settlement*, *maturity*, *price*, *redemption*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

price
specifies the price of security per $100 face value.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: DISC

The following example computes the discount rate for a security.

Statements	Results
`select FINANCE('DISC', mdy(1, 25, 2007), mdy(6, 15, 2007), 97.975, 100, 1);`	`0.05242`

FINANCE DOLLARDE Function

Converts a dollar price, expressed as a fraction, to a dollar price, expressed as a decimal number.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('DOLLARDE', *fractional-dollar*, *fraction*);

Arguments

fractional-dollar
specifies the number expressed as a fraction.

Data type DOUBLE

fraction
specifies the whole number to use in the denominator of a fraction.

Data type DOUBLE

Example: Computing Description: DOLLARDE

The following example converts a dollar price, expressed as a fraction, to a dollar price, expressed as a decimal number.

Statements	Results
`select FINANCE('DOLLARDE', 1.125, 16);`	`1.78125`

FINANCE DOLLARFR Function

Converts a dollar price, expressed as a decimal number, to a dollar price, expressed as a fraction.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('DOLLARFR', *decimal-dollar*, *fraction*);

Arguments

decimal-dollar
specifies a decimal number.

Data type DOUBLE

fraction
specifies the whole number to use in the denominator of a fraction.

Data type DOUBLE

Example: Computing Description: DOLLARFR

The following example converts a dollar price, expressed as a decimal number, to a dollar price, expressed as a fraction.

Statements	Results
`select FINANCE('DOLLARFR', 1.125, 16);`	`1.02`

FINANCE DURATION Function

Computes the annual duration of a security with periodic interest payments.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

FINANCE('DURATION', *settlement*, *maturity*, *coupon*, *yield*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

coupon
specifies the annual coupon rate of the security.

Requirement *Coupon* is provided as a numeric value and not as a percentage.

Data type DOUBLE

yield
specifies the annual yield of the security.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis

specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: DURATION

The following example computes the annual duration of a security with periodic interest payments.

Statements	Results
`select FINANCE('DURATION', mdy(1, 1, 2008), mdy(1, 1, 2016), 0.08, 0.09, 2, 1);`	`5.993775`

FINANCE EFFECT Function

Computes the effective annual interest rate.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('EFFECT', *nominal-rate*, *npery*);

Arguments

nominal-rate
specifies the nominal interest rate.

Data type DOUBLE

npery
specifies the number of compounding periods per year.

Data type DOUBLE

Example: Computing Description: EFFECT

The following example computes the effective annual interest rate.

Statements	Results
`select FINANCE('EFFECT', 0.0525, 4);`	`0.053543`

FINANCE FV Function

Computes the future value of an investment.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('FV', *rate*, *nper*, [*payment*], [*present-value*], [*type*]);

Arguments

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

nper
specifies the total number of payment periods.

Data type DOUBLE

payment
specifies the payment that is made each period; the payment cannot change over the life of the annuity. Typically, *payment* contains principal and interest but no fees and taxes. If *payment* is omitted, you must include the *present-value* argument.

Data type DOUBLE

present-value
specifies the present value or the lump-sum amount that a series of future payments is worth currently. If *present-value* is omitted, it is assumed to be 0 (zero), and you must include the *payment* argument.

Data type DOUBLE

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

If payments are due at the end of the period, then either omit the *type* argument or set it to 0. If payments are due at the beginning of the period, then set *type* to 1.

Data type DOUBLE

Example: Computing Description: FV

The following example computes the future value of an investment.

Statements	Results
`select finance('FV', 0.06/12, 10, -200, -500, 1);`	`2581.403`

FINANCE FVSCHEDULE Function

Computes the future value of the initial principal after applying a series of compound interest rates.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('FVSCHEDULE', *principal*, *schedule-1*, *schedule-2* ...);

Arguments

principal
specifies the present value.

Data type DOUBLE

schedule
specifies the sequence of interest rates to apply.

Requirement *Schedule* rates are provided as numeric values and not as percentages.

Data type DOUBLE

Example: Computing Description: FVSCHEDULE

The following example computes the future value of the initial principal after applying a series of compound interest rates.

Statements	Results
`select FINANCE('FVSCHEDULE', 1, 0.09, 0.11, 0.1);`	`1.33089`

FINANCE INTRATE Function

Computes the interest rate for a fully invested security.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('INTRATE', *settlement*, *maturity*, *investment*, *redemption*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

investment
specifies the amount that is invested in the security.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: INTRATE

The following example computes the interest rate for a fully invested security.

Statements	Results
`select finance('intrate', mdy(2, 15, 2008), mdy(5, 15, 2008), 1000000, 1014420, 2);`	`0.05768`

FINANCE IPMT Function

Computes the interest payment for an investment for a specified period.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('IPMT', *rate*, *period*, *nper*, *pv*, [*fv*], [*type*]);

Arguments

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

period
specifies the period for which you want to calculate the depreciation. *Period* must use the same units as *life*.

Data type DOUBLE

nper
specifies the total number of payment periods.

Data type DOUBLE

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently. If *pv* is omitted, it is assumed to be 0 (zero), and you must include the *fv* argument.

Data type DOUBLE

fv

specifies the future value or a cash balance that you want to attain after the last payment is made. If *fv* is omitted, it is assumed to be 0 (for example, the future value of a loan is 0).

Data type DOUBLE

type

specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

If payments are due at the end of the period, then either omit the *type* argument or set it to 0. If payments are due at the beginning of the period, then set *type* to 1.

Data type DOUBLE

Example: Computing Description: IPMT

The following example computes the interest payment for an investment for a specified period.

Statements	Results
`select FINANCE('IPMT', 0.1/12, 2, 3, 100, ., .);`	`-0.55786`

FINANCE IRR Function

Computes the internal rate of return for a series of cash flows.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('IRR', *value-1*, *value-2*, …, *value-n*);

Arguments

value

specifies a list of numeric arguments that contain numbers for which you want to calculate the internal rate of return.

Data type DOUBLE

Example: Computing Description: IRR

The following example computes the internal rate of return for a series of cash flows.

Statements	Results
`select FINANCE('IRR', -70000, 12000, 15000, 18000, 21000, 26000);`	`0.086631`

FINANCE ISPMT Function

Calculates the interest paid during a specific period of an investment.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('ISPMT', *interest-rate*, *period*, *number-payments*, *pv*);

Arguments

interest-rate
specifies the interest rate for the investment.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

period
specifies the period for which you want to calculate the interest rate. *Period* must be a value between 1 and *number-payments*.

Data type DOUBLE

number-payments
specifies the number of payments for the annuity.

Data type DOUBLE

pv
specifies the loan amount or present value of the payments.

Data type DOUBLE

Example: Computing Description: ISPMT

The following example computes the interest payment for a $5,000 investment that earns 7.5% annually for two years. The interest payment is calculated for the eighth month.

Statements	Results
`select FINANCE('ISPMT', 0.075/12, 8, 2*12, 5000);`	`-20.8333`

FINANCE MDURATION Function

Computes the Macaulay modified duration for a security with an assumed face value of $100.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('MDURATION', *settlement*, *maturity*, *coupon*, *yield*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

coupon
specifies the annual coupon rate of the security.

Requirement *Coupon* is provided as a numeric value and not as a percentage.

Data type DOUBLE

yield
specifies the annual yield of the security.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: MDURATION

The following example computes the Macaulay modified duration for a security with an assumed face value of $100.

Statements	Results
`select FINANCE('MDURATION', mdy(1, 1, 2008), mdy(1, 1, 2016), 0.08, 0.09, 2, 1);`	`5.73567`

FINANCE MIRR Function

Computes the internal rate of return where positive and negative cash flows are financed at different rates.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

FINANCE('MIRR', *value-1*, …, *value-n*, *finance-rate*, *reinvest-rate*);

Arguments

value

specifies a list of numeric arguments that contain numbers. These numbers represent a series of payments (negative values) and income (positive values) that occur at regular periods. *Value* must contain at least one positive value and one negative value to calculate the modified internal rate of return.

Data type DOUBLE

finance-rate

specifies the interest rate that you pay on the money that is used in the cash flows.

Requirement *Finance-rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

reinvest-rate

specifies the interest rate that you receive on the cash flows as you reinvest them.

Requirement *Reinvest-rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

Example: Computing Description: MIRR

The following example computes the internal rate of return where positive and negative cash flows are financed at different rates.

Statements	Results
`select FINANCE('MIRR', -1000, 3000, 4000, 5000, 0.08, 0.10);`	`1.353142`

FINANCE NOMINAL Function

Computes the annual nominal interest rates.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('NOMINAL', *effective-rate*, *npery*);

Arguments

effective-rate
specifies the effective interest rate.

Requirement *Effective-rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

npery
specifies the number of compounding periods per year.

Data type DOUBLE

Example: Computing Description: NOMINAL

The following example computes the annual nominal interest rate.

Statements	Results
`select FINANCE('NOMINAL', 0.08, 4);`	`0.077706`

FINANCE NPER Function

Computes the number of periods for an investment.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

FINANCE('NPER', *rate*, *payment*, *pv*, [*fv*], [*type*]);

Arguments

rate

specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

payment

specifies the payment that is made each period; the payment cannot change over the life of the annuity. Typically, *payment* contains principal and interest but no other fees or taxes. If *payment* is omitted, you must include the *pv* argument.

Data type DOUBLE

pv

specifies the present value or the lump-sum amount that a series of future payments is worth currently. If *pv* is omitted, it is assumed to be 0 (zero), and you must include the *payment* argument.

Data type DOUBLE

fv

specifies the future value or a cash balance that you want to attain after the last payment is made. If *fv* is omitted, it is assumed to be 0 (for example, the future value of a loan is 0).

Data type DOUBLE

type

specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

If payments are due at the end of the period, then either omit the *type* argument or set it to 0. If payments are due at the beginning of the period, then set *type* to 1.

Data type DOUBLE

Example: Computing Description: NPER

The following example computes the number of periods for an investment.

Statements	Results
`select FINANCE('NPER', 0.08, 200, 1000, 0, 0);`	`-4.37198`

FINANCE NPV Function

Computes the net present value of an investment based on a series of periodic cash flows and a discount rate.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('NPV', *rate*, *value-1*, [, … *value-n*]);

Arguments

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

value
represents the sequence of the cash flows.

Data type DOUBLE

Example: Computing Description: NPV

The following example computes the net present value of an investment based on a series of periodic cash flows and a discount rate.

Statements	Results
`select FINANCE('NPV', 0.08, 200, 1000, 0);`	`1042.524`

FINANCE ODDFPRICE Function

Computes the price of a security per $100 face value with an odd first period.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('ODDFPRICE', *settlement*, *maturity*, *issue*, *first-coupon*, *rate*, *yield*, *redemption*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

issue
specifies the issue date of the security.

Requirement *Issue* is a SAS date.

Data type DOUBLE

first-coupon
specifies the first coupon date of the security.

Requirement *First-coupon* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

yield
specifies the annual yield of the security.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: ODDFPRICE

The following example computes the price of a security per $100 face value with an odd first period.

Statements	Results
`select FINANCE('ODDFPRICE', mdy(1, 15, 93), mdy(1, 1, 98), mdy(1, 1, 93), mdy(7, 1, 94), 0.07, 0.06, 100, 2, 0);`	`103.941`

FINANCE ODDFYIELD Function

Computes the yield of a security with an odd first period.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('ODDFYIELD', *settlement*, *maturity*, *issue*, *first-coupon*, *rate*, *price*, *redemption*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

issue
specifies the issue date of the security.

Requirement *Issue* is a SAS date.

Data type DOUBLE

first-coupon
specifies the first coupon date of the security.

Requirement *First-coupon* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

price
specifies the price of the security per $100 face value.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: ODDFYIELD

The following example computes the interest of a yield with an odd first period.

Statements	Results
`select FINANCE('ODDFYIELD', mdy(1, 15, 93), mdy(1, 1, 98), mdy(1, 1, 93), mdy(7, 1, 94), 0.07, 103.94103984, 100, 2, 0);`	`0.06`

FINANCE ODDLPRICE Function

Computes the price of a security per $100 face value with an odd last period.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('ODDLPRICE', *settlement*, *maturity*, *last-interest*, *rate*, *yield*, *redemption*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

last-interest
specifies the last coupon date of the security.

Requirement *Last-interest* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

yield
specifies the annual yield of the security.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: ODDLPRICE

The following example computes the price of a security per $100 face value with an odd last period.

Statements	Results
`select FINANCE('ODDLPRICE', mdy(2, 7, 2008), mdy(6, 15, 2008), mdy(10, 15, 2007), 0.0375, 0.0405, 100, 2, 0);`	`99.87829`

FINANCE ODDLYIELD Function

Computes the yield of a security with an odd last period.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('ODDLYIELD', *settlement*, *maturity*, *last-interest*, *rate*, *price*, *redemption*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

last-interest
specifies the last coupon date of the security.

Requirement *Last-interest* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

price
specifies the price of the security per $100 face value.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: ODDLYIELD

The following example computes the yield of a security with an odd last period.

Statements	Results
`select FINANCE('ODDLYIELD', mdy(2, 7, 2008), mdy(6, 15, 2008), mdy(10, 15, 2007), 0.0375, 99.878286015, 100, 2, 0);`	`0.0405`

FINANCE PMT Function

Computes the periodic payment of an annuity.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

FINANCE('PMT', *rate*, *nper*, *pv*, [*fv*], [*type*]);

Arguments

rate

specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

nper

specifies the total number of payment periods.

Data type DOUBLE

pv

specifies the present value or the lump-sum amount that a series of future payments is worth currently. If *pv* is omitted, it is assumed to be 0 (zero), and you must include the *fv* argument.

Data type DOUBLE

fv

specifies the future value or a cash balance that you want to attain after the last payment is made. If *fv* is omitted, it is assumed to be 0 (for example, the future value of a loan is 0).

Data type DOUBLE

type

specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

If payments are due at the end of the period, then either omit the *type* argument or set it to 0. If payments are due at the beginning of the period, then set *type* to 1.

Data type DOUBLE

Example: Computing Description: PMT

The following example computes the periodic payment for an annuity.

Statements	Results
`select FINANCE('PMT', 0.08, 5, 91, 3, 0);`	`-23.3029`

FINANCE PPMT Function

Computes the payment on the principal for an investment for a specified period.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('PPMT', *rate*, *period*, *nper*, *pv*, [*fv*], [*type*]);

Arguments

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

period
specifies the period.

Range: 1–*nper*

Data type DOUBLE

nper
specifies the number of payment periods.

Data type DOUBLE

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently. If *pv* is omitted, it is assumed to be 0 (zero), and you must include the *fv* argument.

Data type DOUBLE

fv

specifies the future value or a cash balance that you want to attain after the last payment is made. If *fv* is omitted, it is assumed to be 0 (for example, the future value of a loan is 0).

Data type DOUBLE

type

specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

If payments are due at the end of the period, then either omit the *type* argument or set it to 0. If payments are due at the beginning of the period, then set *type* to 1.

Data type DOUBLE

Example: Computing Description: PPMT

The following example computes the payment on the principal for an investment for a specified period.

Statements	Results
`select FINANCE('PPMT', 0.08, 10, 10, 200000, 0, 0);`	`-27598.1`

FINANCE PRICE Function

Computes the price of a security per $100 face value that pays periodic interest.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('PRICE', *settlement*, *maturity*, *rate*, *yield*, *redemption*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

yield
specifies the annual yield of the security.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: PRICE

The following example computes the price of a security per $100 face value that pays periodic interest.

Statements	Results
`select FINANCE('PRICE', mdy(2,15,2008), mdy(11,15,2017), 0.0575, 0.065, 100, 2, 0);`	`94.63436`

FINANCE PRICEDISC Function

Computes the price of a discounted security per $100 face value.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('PRICEDISC', *settlement*, *maturity*, *discount*, *redemption*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

discount
specifies the discount rate of the security.

Requirement *Discount* is provided as a numeric value and not as a percentage.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: PRICEDISC

The following example computes the price of a discounted security per $100 face value.

Statements	Results
`select FINANCE('PRICEDISC', mdy(2,15,2008), mdy(11,15,2017), 0.0525, 100, 0);`	`48.8125`

FINANCE PRICEMAT Function

Computes the price of a security per $100 face value that pays interest at maturity.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('PRICEMAT', *settlement*, *maturity*, *issue*, *rate*, *yield*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

issue
specifies the issue date of the security.

Requirement *Issue* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

yield
specifies the annual yield of the security.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360

Numeric Value	String Value	Day Count Method
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: PRICEMAT

The following example computes the price of a security per $100 face value that pays interest at maturity.

Statements	Results
`select FINANCE('PRICEMAT', mdy(2,15,2008), mdy(4, 13, 2008), mdy(11, 11, 2007), 0.061, 0.061, 0);`	`99.9845`

FINANCE PV Function

Computes the present value of an investment.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('PV', *rate*, *nper*, *payment*, [*fv*], [*type*]);

Arguments

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

nper
specifies the total number of payment periods.

Data type DOUBLE

payment
specifies the payment that is made each period; the payment cannot change over the life of the annuity. Typically, *payment* contains principal and interest but no other fees or taxes.

Data type DOUBLE

fv
specifies the future value or a cash balance that you want to attain after the last payment is made. If *fv* is omitted, it is assumed to be 0 (for example, the future value of a loan is 0).

Data type DOUBLE

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

If payments are due at the end of the period, then either omit the *type* argument or set it to 0. If payments are due at the beginning of the period, then set *type* to 1.

Data type DOUBLE

Example: Computing Description: PV

The following example computes the present value of an investment.

Statements	Results
`select FINANCE('PV', 0.05, 10, 1000, 200, 0);`	`-7844.52`

FINANCE RATE Function

Computes the interest rate per period of an annuity.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('RATE', *nper*, *payment*, *pv*, [*fv*], [*type*]);

Arguments

nper
specifies the total number of payment periods.

Data type DOUBLE

payment
specifies the payment that is made each period; the payment cannot change over the life of the annuity. Typically, *payment* contains principal and interest but no other fees or taxes. If *payment* is omitted, you must include the *pv* argument.

Data type DOUBLE

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently. If *pv* is omitted, it is assumed to be 0 (zero), and you must include the *fv* argument.

Data type DOUBLE

fv
specifies the future value or a cash balance that you want to attain after the last payment is made. If *fv* is omitted, it is assumed to be 0 (for example, the future value of a loan is 0).

Data type DOUBLE

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

If payments are due at the end of the period, then either omit the *type* argument or set it to 0. If payments are due at the beginning of the period, then set *type* to 1.

Data type DOUBLE

Example: Computing Description: RATE

The following example computes the interest rate per period of an annuity.

Statements	Results
`select FINANCE('RATE', 4, -2481, 8000);`	`0.092148`

FINANCE RECEIVED Function

Computes the amount that is received at maturity for a fully invested security.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('RECEIVED', *settlement*, *maturity*, *investment*, *discount*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

investment
specifies the amount that is invested in the security.

Data type DOUBLE

discount
specifies the discount rate of the security.

Requirement *Discount* is provided as a numeric value and not as a percentage.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360

Numeric Value	String Value	Day Count Method
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: RECEIVED

The following example computes the amount that is received at maturity for a fully invested security.

Statements	Results
`select FINANCE('RECEIVED', mdy(2,15,2008), mdy(5, 15, 2008), 1000000, 0.0575, 2);`	`1014585`

FINANCE SLN Function

Computes the straight-line depreciation of an asset for one period.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('SLN', *cost*, *salvage*, *life*);

Arguments

cost
specifies the initial cost of the asset.

Data type DOUBLE

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

Data type DOUBLE

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

Data type DOUBLE

Example: Computing Description: SLN

The following example computes the straight-line depreciation of an asset for one period.

Statements	Results
`select FINANCE('SLN', 2000, 200, 11);`	`163.6364`

FINANCE SYD Function

Computes the sum-of-years digits depreciation of an asset for a specified period.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('SYD', *cost*, *salvage*, *life*, *period*);

Arguments

cost
specifies the initial cost of the asset.

Data type DOUBLE

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

Data type DOUBLE

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

Data type DOUBLE

period
specifies a period in the same time units that are used for the argument *life*.

Data type DOUBLE

Example: Computing Description: SYD

The following example computes the sum-of-years digits depreciation of an asset for a specified period.

Statements	Results
`select FINANCE('SYD', 2000, 200, 11, 1);`	`300`

FINANCE TBILLEQ Function

Computes the bond-equivalent yield for a treasury bill.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('TBILLEQ', *settlement*, *maturity*, *discount*);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

discount
specifies the discount rate of the security.

Requirement *Discount* is provided as a numeric value and not as a percentage.

Data type DOUBLE

Example: Computing Description: TBILLEQ

The following example computes the bond-equivalent yield for a treasury bill.

Statements	Results
`select FINANCE('TBILLEQ', mdy(3, 31, 2008), mdy(6, 1, 2008), 0.0914);`	`0.094151`

FINANCE TBILLPRICE Function

Computes the price of a treasury bill per $100 face value.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('TBILLPRICE', *settlement*, *maturity*, *discount*);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

discount
specifies the discount rate of the security.

Requirement *Discount* is provided as a numeric value and not as a percentage.

Data type DOUBLE

Example: Computing Description: TBILLPRICE

The following example computes the price of a treasury bill per $100 face value.

Statements	Results
`select FINANCE('TBILLPRICE', mdy(3, 31, 2008), mdy(6, 1, 2008), 0.09);`	`98.45`

FINANCE TBILLYIELD Function

Computes the yield for a treasury bill.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('TBILLYIELD', *settlement*, *maturity*, *price*);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

price
specifies the price of the security per $100 face value.

Data type DOUBLE

Example: Computing Description: TBILLYIELD

The following example computes the yield for a treasury bill.

Statements	Results
`select FINANCE('TBILLYIELD', mdy(3, 31, 2008), mdy(6, 1, 2008), 98);`	`0.118499`

FINANCE VDB Function

Computes the depreciation of an asset for a specified or partial period by using a declining balance method.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('VDB', *cost*, *salvage*, *life*, *start-period*, *end-period*, [*factor*], [*noswitch*]);

Arguments

cost
specifies the initial cost of the asset.

Data type DOUBLE

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

Data type DOUBLE

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

Data type DOUBLE

start-period
specifies the first period in the calculation. Payment periods are numbered beginning with 1.

Data type DOUBLE

end-period
specifies the last period in the calculation.

Data type DOUBLE

factor
specifies the rate at which the balance declines. If *factor* is omitted, it is assumed to be 2 (the double-declining balance method).

Data type DOUBLE

noswitch
specifies a logical value that determines whether to switch to straight-line depreciation when the depreciation is greater than the declining balance calculation. If *noswitch* is omitted, it is assumed to be 1.

Data type DOUBLE

Example: Computing Description: VDB

The following example computes the depreciation of an asset for a specified or partial period by using a declining balance method.

Statements	Results
`select FINANCE('VDB', 2400, 300, 10, 0, 1, 1.5);`	`360`

FINANCE XIRR Function

Computes the internal rate of return for a schedule of cash flows that is not necessarily periodic.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

FINANCE('XIRR', *values*, *dates*, [*guess*]);

Arguments

values

specifies a series of cash flows that corresponds to a schedule of payments in dates. The first payment is optional and corresponds to a cost or payment that occurs at the beginning of the investment. If the first value is a cost or payment, it must be a negative value. All succeeding payments are discounted based on a 365-day year. The series of values must contain at least one positive value and one negative value.

Data type DOUBLE

dates

specifies a schedule of payment dates that corresponds to the cash flow payments. The first payment date indicates the beginning of the schedule of payments. All other dates must be later than this date, but they can occur in any order.

Requirement *Dates* are SAS dates.

Data type DOUBLE

guess

specifies an optional number that you guess is close to the result of XIRR.

Data type DOUBLE

Example: Computing Description: XIRR

The following example computes the internal rate of return for a schedule of cash flows that is not necessarily periodic.

Statements	Results
`select FINANCE('XIRR', -10000, 2750, 4250, 3250, 2750, mdy(1, 1, 2008), mdy(3, 1, 2008), mdy(10, 30, 2008), mdy(2, 15, 2009), mdy(4, 1, 2009), 0.1);`	`0.373363`

FINANCE XNPV Function

Computes the net present value for a schedule of cash flows that is not necessarily periodic.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('XNPV', *rate*, *values*, *dates*);

Arguments

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

values
specifies a series of cash flows that corresponds to a schedule of payments in dates. The first payment is optional and corresponds to a cost or payment that occurs at the beginning of the investment. If the first value is a cost or payment, it must be a negative value. All succeeding payments are discounted based on a 365-day year. The series of values must contain at least one positive value and one negative value.

Data type DOUBLE

dates
specifies a schedule of payment dates that corresponds to the cash flow payments. The first payment date indicates the beginning of the schedule of payments. All other dates must be later than this date, but they can occur in any order.

Requirement *Dates* are SAS dates.

Data type DOUBLE

Example: Computing Description: XNPV

The following example computes the net present value for a schedule of cash flows that is not necessarily periodic.

Statements	Results
`select FINANCE('XNPV', .09, -10000, 2750, 4250, 3250, 2750, mdy(1, 1, 2008), mdy(3, 1, 2008), mdy(10, 30, 2008), mdy(2, 15, 2009), mdy(4, 1, 2009));`	`2086.648`

FINANCE YIELD Function

Computes the yield on a security that pays periodic interest.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('YIELD', *settlement*, *maturity*, *rate*, *price*, *redemption*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

price
specifies the price of the security per $100 face value.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: YIELD

The following example computes the yield on a security that pays periodic interest.

Statements	Results
`select FINANCE('YIELD', mdy(2, 15, 2008), mdy(11, 15, 2016), 0.0575, 95.04287, 100, 2, 0);`	`0.065`

FINANCE YIELDDISC Function

Computes the annual yield for a discounted security (for example, a treasury bill).

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('YIELDDISC', *settlement*, *maturity*, *price*, *redemption*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

price
specifies the price of the security per $100 face value.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual

Numeric Value	String Value	Day Count Method
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: YIELDDISC

The following example computes the annual yield for a discounted security (for example, a treasury bill).

Statements	Results
`select FINANCE('YIELDDISC', mdy(2,15,2008), mdy(11, 15, 2016), 95.04287, 100, 0);`	`0.005961`

FINANCE YIELDMAT Function

Computes the annual yield of a security that pays interest at maturity.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('YIELDMAT', *settlement*, *maturity*, *issue*, *rate*, *price*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

issue
specifies the issue date of the security.

Requirement *Issue* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

price
specifies the price of the security per $100 face value.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: YIELDMAT

The following example computes the annual yield of a security that pays interest at maturity.

Statements	Results
`select FINANCE('YIELDMAT', mdy(3, 15, 2008), mdy(11, 3, 2008), mdy(11, 8, 2007), 0.0625, 100.0123, 0);`	`0.060954`

FLOOR Function

Returns the largest integer less than or equal to a numeric value expression.

Categories: CAS
Truncation

Returned data type: DECIMAL, DOUBLE, NUMERIC

Syntax

FLOOR(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DECIMAL, DOUBLE, NUMERIC

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

If *expression* is within 1E-12 of an integer, the function returns that integer. If the result is a number that does not fit into the range of a DOUBLE, the FLOOR function fails.

If the argument is DECIMAL, the result is DECIMAL. Otherwise, the argument is converted to DOUBLE (if not so already), and the result is DOUBLE.

Comparisons

The FLOOR function fuzzes the results so that if the results are within 1E-12 of an integer, the FLOOR function returns that integer. The FLOORZ function uses zero fuzzing. Therefore, with the FLOORZ function, you might get unexpected results.

Example

Table: DENSITIES on page 1173

The following statement illustrates the FLOOR function:

Statements	Results
`select floor(1.95);`	`1`

`select floor(density) from densities;`	`67`
	`306`
	`30`
	`323`
	`20`
	`383`
	`31`
	`309`
	`6`
	`247`

See Also

Functions:

- “CEIL Function” on page 358
- “CEILZ Function” on page 359
- “FLOORZ Function” on page 521

FLOORZ Function

Returns the largest integer that is less than or equal to the argument, using zero fuzzing.

Categories: CAS
Truncation

Returned data type: DOUBLE

Syntax

FLOORZ(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Comparisons

Unlike the FLOOR function, the FLOORZ function uses zero fuzzing. If the argument is within 1E-12 of an integer, the FLOOR function fuzzes the result to be equal to that integer. The FLOORZ function does not fuzz the result. Therefore, with the FLOORZ function, you might get unexpected results.

Example

Table: DENSITIES on page 1173

The following statements illustrate the FLOORZ function:

Statements	Results
`select floorz(-2.4);`	`-3`
`select floorz(-1.6);`	`-2`

`select floorz(density) from densities;`	`67`
	`306`
	`30`
	`323`
	`20`
	`383`
	`31`
	`309`
	`6`
	`247`

See Also

Functions:

- "CEIL Function" on page 358
- "CEILZ Function" on page 359
- "FLOOR Function" on page 520

FMTINFO Function

Returns information about a SAS format or informat.

Categories: CAS
Special

Restriction: This function returns information about formats that are supplied by SAS. It cannot be used for user-defined formats that are created with the FORMAT procedure.

Syntax

FMTINFO('*format-name*', '*information-type*');

Arguments

'*format-name*'
specifies the name of a SAS format or informat.

Requirement *format-name* must be enclosed in single quotation marks.

information-type
specifies the type of information that is returned. *format-information* can be one of the following values:

'CAT'
returns the function category.

'TYPE'
returns whether the *format-name* is a format, an informat, or both.

'DESC'
returns a short description of the format or informat.

'MIND'
returns the minimum number of digits to the right of the decimal place in the format or informat.

'MAXD'
returns the maximum number of digits to the right of the decimal place in the format or informat.

'DEFD'
returns the default number of digits to the right of the decimal place in the format or informat.

'MINW'
returns the minimum width value of the format or informat.

'MAXW'
returns the maximum width value of the format or informat.

'DEFW'
returns the default width value of the format or informat.

Restriction You can specify only one *information-type* argument.

Requirement *information-type* must be enclosed in single quotation marks.

Details

The FMTINFO function returns information about a format or informat. You can return information about a format or informat's category, the type of language element, a description of the language element, and the minimum, maximum, and default decimal and width values.

You cannot specify multiple arguments with the FMTINFO function.

The FMTINFO function returns a character string for all data values, including the numeric value arguments MIND, MAXD, DEFD, MINW, MAXW, and DEFW.

Example

The following example returns information about the DATE format.

```
select fmtinfo('date','type') as ftype,
       fmtinfo('date','cat') as fcat,
       fmtinfo('date','desc') as fdesc,
       fmtinfo('date','mind') as fmind,
       fmtinfo('date','maxd') as fmaxd,
       fmtinfo('date','defd') as fdefd,
       fmtinfo('date','minw') as fminw,
       fmtinfo('date','maxw') as fmaxw,
       fmtinfo('date','defw') as fdefw;
```

Output 4.8 *Result of FMTINFO Query on DATE Format*

FTYPE	FCAT	FDESC	FMIND	FMAXD	FDEFD	FMINW	FMAXW	FDEFW
BOTH	date	date value	0	8	0	5	11	7

FNONCT Function

Returns the value of the noncentrality parameter of an F distribution.

Categories:	CAS Mathematical
Returned data type:	DOUBLE

Syntax

FNONCT(*x*, *ndf*, *ddf*, *probability*)

Arguments

x
is a numeric random variable.

Range $x \geq 0$

Data type DOUBLE

ndf
is a numeric numerator degree of freedom parameter.

Range $ndf > 0$

Data type DOUBLE

ddf
is a numeric denominator degree of freedom parameter.

Range $ddf > 0$

Data type DOUBLE

probability
is a probability.

Range $0 < probability < 1$

Data type DOUBLE

Details

The FNONCT function returns the nonnegative noncentrality parameter from a noncentral *F* distribution whose parameters are *x*, *ndf*, *ddf*, and *nc*. If *probability* is

greater than the probability from the central *F* distribution whose parameters are *x*, *ndf*, and *ddf*, a root to this problem does not exist. In this case a missing value is returned. A Newton-type algorithm is used to find a nonnegative root *nc* of the equation

$$P_f(x|ndf, ddf, nc) - prob = 0$$

The following relationship applies to the preceding equation:

$$P_f(x|ndf, ddf, nc) = \varepsilon^{\frac{-nc}{2}} \sum_{j=0}^{\infty} \frac{\left(\frac{nc}{2}\right)^j}{j!} I_{\frac{(ndf)x}{ddf+(ndf)x}}\left(\frac{ddf}{2} + j, \frac{ddf}{2}\right)$$

In the equation, *I* (. . .) is the probability from the beta distribution that is given by the following equation:

$$I_x(a,b) = \frac{}{\frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}} \int_0^x t^{a-1}(1-t)^{b-1} dt$$

If the algorithm fails to converge to a fixed point, a missing value is returned.

Example

The following example computes the noncentrality parameter from the *F* distribution.

```
proc ds2;
data test /overwrite=yes;
   dcl double x df ddf nc prob;
   method init();
      x=2;
      df=4;
      ddf=5;
      do nc=1 to 3 by .5;
         prob=probf(x, df, ddf, nc);
         output;
      end;
   end;
enddata;
run;
quit;

proc print data=test;
run;

proc fedsql;
 select * from test;
 select fnonct(x, df, ddf, prob) from test;
quit;
```

Output 4.9 *Contents of Table Test*

x	df	ddf	nc	prob
2	4	5	1	0.692767
2	4	5	1.5	0.657008
2	4	5	2	0.622321
2	4	5	2.5	0.588776
2	4	5	3	0.556424

Output 4.10 *Result of FNONCT Function on Table Test*

FNONCT
1
1.5
2
2.5
3

See Also

Functions:

- "CNONCT Function" on page 362
- "TNONCT Function" on page 836

FUZZ Function

Returns the nearest whole number if the argument is within 1E-12 of that number.

Categories:	CAS Truncation
Returned data type:	DOUBLE

Syntax

FUZZ(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The FUZZ function returns the nearest whole number if the expression is within 1E-12 of the number (that is, if the absolute difference between the whole number and argument is less than 1E-12). Otherwise, the expression is returned.

Example

The following statement illustrates the FUZZ function:

Statements	Results
`select fuzz(5.99999999);`	`5.99999999`

GAMINV Function

Returns a quantile from the gamma distribution.

Categories: CAS
Quantile

Returned data type: DOUBLE

Syntax

GAMINV(*p*, *a*)

Arguments

p

specifies any valid expression that evaluates to a numeric probability.

Range $0 \le p < 1$

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

a

specifies any valid expression that evaluates to a numeric shape parameter.

Range $a > 0$

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The GAMINV function returns the pth quantile from the gamma distribution, with shape parameter *a*. The probability that a row from a gamma distribution is less than or equal to the returned quantile is *p*.

Note: GAMINV is the inverse of the PROBGAM function.

Example

The following statements illustrate the GAMINV function:

Statement	Result
`select gaminv(0.5, 9);`	`8.6689511844`
`select gaminv(0.1, 2.1);`	`0.5841932369`

See Also

Functions:

- "PROBGAM Function" on page 750

GAMMA Function

Returns the value of the gamma function.

Categories:	CAS Mathematical
Returned data type:	DOUBLE

Syntax

GAMMA(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Restriction	Nonpositive integers are invalid.
Data type	DOUBLE
See	"<sql-expression> Expression" on page 894 "FedSQL Expressions" on page 46

Details

The GAMMA function returns the integral, which is given by the following equation.

$$\mathrm{GAMMA}(x) = \int_0^\infty t^{x-1} e^{-t}\, dt.$$

For positive integers, GAMMA(x) is (x − 1)!. This function is commonly denoted by $\Gamma(x)$.

Example

The following statement illustrates the GAMMA function:

Statements	Results
`select gamma(6);`	`120`

GARKHCLPRC Function

Calculates call prices for European options on stocks, based on the Garman-Kohlhagen model.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

GARKHCLPRC(*E*, *t*, *S*, R_d, R_f, *sigma*)

Arguments

E

is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

Data type DOUBLE

t

is a nonmissing value that specifies the time to maturity.

Data type DOUBLE

S

is a nonmissing, positive value that specifies the spot currency price.

Requirement Specify *S* and *E* in the same units.

Data type DOUBLE

R_d

is a nonmissing, positive fraction that specifies the risk-free domestic interest rate for period *t*.

Requirement Specify a value for R_d for the same time period as the unit of *t*.

Data type DOUBLE

R_f
is a nonmissing, positive fraction that specifies the risk-free foreign interest rate for period *t*.

Requirement Specify a value for R_f for the same time period as the unit of *t*.

Data type DOUBLE

sigma
is a nonmissing, positive fraction that specifies the volatility of the currency rate.

Requirement Specify a value for *sigma* for the same time period as the unit of *t*.

Data type DOUBLE

Details

The GARKHCLPRC function calculates the call prices for European options on stocks, based on the Garman-Kohlhagen model. The function is based on the following relationship:

$$\text{CALL} = SN(d_1)\left(\varepsilon^{-R_f t}\right) - EN(d_2)\left(\varepsilon^{-R_d t}\right)$$

Arguments

S
specifies the spot currency price.

N
specifies the cumulative normal density function.

E
specifies the exercise price of the option.

t
specifies the time to expiration.

R_d
specifies the risk-free domestic interest rate for period *t*.

R_f
specifies the risk-free foreign interest rate for period *t*.

$$d_1 = \frac{\left(\ln\left(\frac{S}{E}\right) + \left(R_d - R_f + \frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

The following arguments apply to the preceding equation:

σ
specifies the volatility of the underlying asset.

σ^2
specifies the variance of the rate of return.

For the special case of *t*=0, the following equation is true:

$$\text{CALL} = \max((S - E), 0)$$

For information about the basics of pricing, see "Using Pricing Functions" in *SAS Functions and CALL Routines: Reference*.

Comparisons

The GARKHCLPRC function calculates the call prices for European options on stocks, based on the Garman-Kohlhagen model. The GARKHPTPRC function calculates the put prices for European options on stocks, based on the Garman-Kohlhagen model. These functions return a scalar value.

Example

The following statements illustrate the GARKHCLPRC function:

Statements	Results
	`----+----1----+---2--`
`select garkhclprc(40, .5, 38, .06, .04, .2);`	`1.449425`
`select garkhclprc(19, .25, 20, .05, .03, .09);`	`1.130421`

See Also

Functions:

- "GARKHPTPRC Function" on page 533

GARKHPTPRC Function

Calculates put prices for European options on stocks, based on the Garman-Kohlhagen model.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

GARKHPTPRC(*E*, *t*, *S*, R_d, R_f, *sigma*)

Arguments

E

is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

Data type DOUBLE

t

is a nonmissing value that specifies the time to maturity, in years.

Data type DOUBLE

S

is a nonmissing, positive value that specifies the spot currency price.

Requirement Specify *S* and *E* in the same units.

Data type DOUBLE

R_d

is a nonmissing, positive fraction that specifies the risk-free domestic interest rate for period *t*.

Requirement Specify a value for R_d for the same time period as the unit of *t*.

Data type DOUBLE

R_f

is a nonmissing, positive fraction that specifies the risk-free foreign interest rate for period *t*.

Requirement Specify a value for R_t for the same time period as the unit of *t*.

Data type DOUBLE

sigma

is a nonmissing, positive fraction that specifies the volatility of the currency rate.

Data type DOUBLE

Details

The GARKHPTPRC function calculates the put prices for European options on stocks, based on the Garman-Kohlhagen model. The function is based on the following relationship:

$$\text{PUT} = \text{CALL} - S\left(\varepsilon^{-R_f t}\right) + E\left(\varepsilon^{-R_d t}\right)$$

Arguments

S
specifies the spot currency price.

E
specifies the exercise price of the option.

t
specifies the time to expiration, in years.

R_d
specifies the risk-free domestic interest rate for period *t*.

R_f
specifies the risk-free foreign interest rate for period *t*.

$$d_1 = \frac{\left(\ln\left(\frac{S}{E}\right) + \left(R_d - R_f + \frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

The following arguments apply to the preceding equation:

σ
specifies the volatility of the underlying asset.

σ^2
specifies the variance of the rate of return.

For the special case of *t*=0, the following equation is true:

$$\text{PUT} = \max((E - S), 0)$$

For information about the basics of pricing, see "Using Pricing Functions" in *SAS Functions and CALL Routines: Reference*.

Comparisons

The GARKHPTPRC function calculates the put prices for European options on stocks, based on the Garman-Kohlhagen model. The GARKHCLPRC function calculates the call prices for European options on stocks, based on the Garman-Kohlhagen model. These functions return a scalar value.

Example

The following statements illustrate the GARKHPTPRC function:

Statements	Results
	`----+----1----+---2--`
`select garkhptprc(50, .7, 55, .05, .04, .2);`	`1.405088`

Statements	Results
`select garkhptprc(32, .3, 33, .05, .03, .3);`	`1.564732`

See Also

Functions:

- "GARKHCLPRC Function" on page 531

GCD Function

Returns the greatest common divisor for a set of integers.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

GCD(*expression-1*, *expression-2* [, …*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement At least two arguments are required.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The GCD (greatest common divisor) function returns the greatest common divisor of one or more integers. For example, the greatest common divisor for 30 and 42 is 6. The greatest common divisor is also called the highest common factor.

Example

The following statements illustrate the GCD function:

Statements	Results
`select gcd(5,15);`	5
`select gcd(36,45);`	9

See Also

Functions:

- "LCM Function" on page 612

GEOMEAN Function

Returns the geometric mean.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

GEOMEAN(*expression* [, …*expression*])

Arguments

expression
is any valid expression that evaluates to a nonnegative numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

If any argument is negative, then the result is a null or missing value. A message appears in the log that the negative argument is invalid. If any argument is zero, then the geometric mean is zero. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the geometric mean of the non-null or nonmissing values.

Let n be the number of arguments with non-null or nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The geometric mean is the nth root of the product of the values:

$$\sqrt[n]{(x_1 * x_2 * \ldots * x_n)}$$

Equivalently, the geometric mean is shown in this equation.

$$\exp\left(\frac{(\log(x_1) + \log(x_2) + \ldots + \log(x_n))}{n}\right)$$

Floating-point arithmetic often produces tiny numerical errors. Some computations that result in zero when exact arithmetic is used might result in a tiny nonzero value when floating-point arithmetic is used. Therefore, GEOMEAN fuzzes the values of arguments that are approximately zero. When the value of one argument is extremely small relative to the largest argument, the former argument is treated as zero. If you do not want SAS to fuzz the extremely small values, then use the GEOMEANZ function.

Comparisons

The MEAN function returns the arithmetic mean (average), and the HARMEAN function returns the harmonic mean, whereas the GEOMEAN function returns the geometric mean of the non-null or nonmissing values. Unlike GEOMEANZ, GEOMEAN fuzzes the values of the arguments that are approximately zero.

Example

The following statements illustrate the GEOMEAN function:

Statements	Results
`select geomean(1,2,2,4);`	2
`select geomean(.,2,4,8);`	4

See Also

Functions:

- "GEOMEANZ Function" on page 539
- "HARMEAN Function" on page 540
- "HARMEANZ Function" on page 542
- "MEAN Function" on page 645

GEOMEANZ Function

Returns the geometric mean, using zero fuzzing.

Categories:	CAS Descriptive Statistics
Returned data type:	DOUBLE

Syntax

GEOMEANZ(*expression* [, …*expression*])

Arguments

expression

specifies any valid expression that evaluates to a nonnegative numeric value.

Data type	DOUBLE
See	"<sql-expression> Expression" on page 894 "FedSQL Expressions" on page 46

Details

If any argument is negative, then the result is a null or missing value. A message appears in the log that the negative argument is invalid. If any argument is zero, then the geometric mean is zero. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the geometric mean of the non-null or nonmissing values.

Let n be the number of arguments with non-null or nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The geometric mean is the nth root of the product of the values:

$$\sqrt[n]{(x_1 * x_2 * \ldots * x_n)}$$

Equivalently, the geometric mean is shown in this equation.

$$\exp\left(\frac{(\log(x_1) + \log(x_2) + \ldots + \log(x_n))}{n}\right)$$

Comparisons

The MEAN function returns the arithmetic mean (average), and the HARMEAN function returns the harmonic mean, whereas the GEOMEANZ function returns the geometric mean of the non-null or nonmissing values. Unlike GEOMEAN, GEOMEANZ does not fuzz the values of the arguments that are approximately zero.

Example

The following statements illustrate the GEOMEANZ function:

Statements	Results
`select geomeanz(1,2,2,4);`	2
`select geomeanz(.,2,4,8);`	4

See Also

Functions:

- “GEOMEAN Function” on page 537
- “HARMEAN Function” on page 540
- “HARMEANZ Function” on page 542
- “MEAN Function” on page 645

HARMEAN Function

Returns the harmonic mean.

Categories:	CAS Descriptive Statistics
Returned data type:	DOUBLE

Syntax

HARMEAN(*expression* [, …*expression*])

Arguments

expression

specifies any valid expression that evaluates to a nonnegative numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

If any argument is negative, then the result is a null or missing value. A message appears in the log that the negative argument is invalid. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the harmonic mean of the non-null or nonmissing values.

If any argument is zero, then the harmonic mean is zero. Otherwise, the harmonic mean is the reciprocal of the arithmetic mean of the reciprocals of the values.

Let n be the number of arguments with non-null or nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The harmonic mean is shown in this equation.

$$\frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n}}$$

Floating-point arithmetic often produces tiny numerical errors. Some computations that result in zero when exact arithmetic is used might result in a tiny nonzero value when floating-point arithmetic is used. Therefore, HARMEAN fuzzes the values of arguments that are approximately zero. When the value of one argument is extremely small relative to the largest argument, the former argument is treated as zero. If you do not want SAS to fuzz the extremely small values, then use the HARMEANZ function.

Comparisons

The MEAN function returns the arithmetic mean (average), and the GEOMEAN function returns the geometric mean, whereas the HARMEAN function returns the harmonic mean of the non-null or nonmissing values. Unlike HARMEANZ, HARMEAN fuzzes the values of the arguments that are approximately zero.

Example

The following statements illustrate the HARMEAN function:

Statements	Results
`select harmean(1,2,4,4);`	2
`select harmean(.,4,12,24);`	8

See Also

Functions:

- "GEOMEAN Function" on page 537
- "GEOMEANZ Function" on page 539
- "HARMEANZ Function" on page 542
- "MEAN Function" on page 645

HARMEANZ Function

Returns the harmonic mean, using zero fuzzing.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

HARMEANZ(*expression* [, …*expression*])

Arguments

expression
specifies any valid expression that evaluates to a nonnegative numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

If any argument is negative, then the result is a null or value. A message appears in the log that the negative argument is invalid. If all the arguments are null or values, then the result is a null or value. Otherwise, the result is the harmonic mean of the non-null or nonmissing values.

If any argument is zero, then the harmonic mean is zero. Otherwise, the harmonic mean is the reciprocal of the arithmetic mean of the reciprocals of the values.

Let n be the number of arguments with non-null or nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The harmonic mean is shown in this equation.

$$\frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n}}$$

Comparisons

The MEAN function returns the arithmetic mean (average), and the GEOMEAN function returns the geometric mean, whereas the HARMEANZ function returns the harmonic mean of the non-null or nonmissing values. Unlike HARMEAN, HARMEANZ does not fuzz the values of the arguments that are approximately zero.

Example

The following statements illustrate the HARMEANZ function:

Statements	Results
`select harmeanz(1,2,4,4);`	2
`select harmeanz(.,4,12,24);`	8

See Also

Functions:

- “GEOMEAN Function” on page 537
- “GEOMEANZ Function” on page 539
- “HARMEAN Function” on page 540
- “MEAN Function” on page 645

HMS Function

Returns a SAS time value from hour, minute, and second values.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

HMS(*hour*, *minute*, *second*)

Arguments

hour
specifies a numeric expression that represents a whole number from 1 through 12.

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

minute
specifies a numeric expression that represents a whole number from 1 through 59.

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

second
specifies a numeric expression that represents a whole number from 1 through 59.

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

Details

The HMS function returns a numeric value that represents a SAS time value. A SAS time value is a number that represents the number of seconds since midnight of the current day.

Example

The following statements illustrate the HMS function:

Statements	Results
`select hms(12,45,10);`	`45910`
`select put(hms(12,45,10), time.);`	`12:45:10`

See Also

Concepts:

- "Dates and Times in FedSQL" on page 58

Functions:

- "DHMS Function" on page 429
- "HOUR Function" on page 549
- "MINUTE Function" on page 649
- "SECOND Function" on page 806

HOLIDAY Function

Returns a SAS date value of a specified holiday for a specified year.

Categories:	CAS Date and Time
Returned data type:	DOUBLE

Syntax

HOLIDAY('*holiday*', *year*)

Arguments

'*holiday*'

is a character constant, variable, or expression that specifies one of the values listed in the following table.

Values for *holiday* can be in uppercase or lowercase.

Table 4.1 *Holiday Values and Their Descriptions*

Holiday Value	Description	Date Celebrated
BOXING	Boxing Day	December 26
CANADA	Canadian Independence Day	July 1
CANADAOBSERVED	Canadian Independence Day observed	July 1, or July 2 if July 1 is a Sunday
CHRISTMAS	Christmas	December 25
COLUMBUS	Columbus Day	2nd Monday in October
EASTER	Easter Sunday	date varies
FATHERS	Father's Day	3rd Sunday in June
HALLOWEEN	Halloween	October 31
LABOR	Labor Day	1st Monday in September
MLK	Martin Luther King, Jr. 's birthday	3rd Monday in January beginning in 1986
MEMORIAL	Memorial Day	last Monday in May (since 1971)
MOTHERS	Mother's Day	2nd Sunday in May
NEWYEAR	New Year's Day	January 1
THANKSGIVING	U.S. Thanksgiving Day	4th Thursday in November

Holiday Value	Description	Date Celebrated
THANKSGIVINGCANADA	Canadian Thanksgiving Day	2nd Monday in October
USINDEPENDENCE	U.S. Independence Day	July 4
USPRESIDENTS	Abraham Lincoln's and George Washington's birthdays observed	3rd Monday in February (since 1971)
VALENTINES	Valentine's Day	February 14
VETERANS	Veterans Day	November 11
VETERANSUSG	Veterans Day - U.S. government-observed	U.S. government-observed date for Monday–Friday schedule
VETERANSUSPS	Veterans Day - U.S. post office observed	U.S. government-observed date for Monday–Saturday schedule (U.S. Post Office)
VICTORIA	Victoria Day	Monday on or preceding May 24

Data type CHAR

year
is a numeric constant, variable, or expression that specifies a four-digit year. If you use a two-digit year, then you must specify the YEARCUTOFF= system option.

Data type DOUBLE

Details

The HOLIDAY function computes the date on which a specific holiday occurs in a specified year. Only certain common U.S. and Canadian holidays are defined for use with this function. (See "Holiday Values and Their Descriptions" in *SAS Functions and CALL Routines: Reference* for a list of valid holidays.)

The definition of many holidays has changed over the years. In the U.S., Executive Order 11582, issued on February 11, 1971, fixed the observance of many U.S. federal holidays.

The current holiday definition is extended indefinitely into the past and future, although many holidays have a fixed date at which they were established. Some holidays have not had a consistent definition in the past.

The HOLIDAY function returns a SAS date value. To convert the SAS date value to a calendar date, use any valid SAS date format, such as the DATE9. format.

Comparisons

In some cases, the HOLIDAY function and the NWKDOM function return the same result. For example, the statement `holiday('thanksgiving', 2012);` returns the same value as `nwkdom(4, 5, 11, 2012);`.

In other cases, the HOLIDAY function and the MDY function return the same result. For example, the statement `holiday('christmas', 2012);` returns the same value as `mdy(12, 25, 2012);`.

Example

The following statements illustrate the HOLIDAY function:

Statements	Results
`select put(holiday('thanksgiving', 2013), date.);`	28NOV2013
`select put(holiday('boxing', 2013), date.);`	26DEC2013
`select put(holiday('easter', 2013), date.);`	31MAR2013
`select put(holiday('canada', 2013), date.);`	01JUL2013
`select put(holiday('fathers', 2013), date.);`	16JUN2013
`select put(holiday('valentines', 2013), date.);`	14FEB2013
`select put(holiday('victoria', 2013), date.);`	20MAY2013

See Also

Functions:

- “MDY Function” on page 644

HOUR Function

Returns the hour from a time or datetime value.

Categories:	Date and Time CAS
Returned data type:	TINYINT

Syntax

HOUR(*time* | *datetime*)

Arguments

time

specifies any valid expression that represents a time value.

Data type TIME

See "FedSQL Expressions" on page 46

datetime

specifies any valid expression that represents a datetime value.

Data type TIMESTAMP

See "FedSQL Expressions" on page 46

Example

Table: CUSTONLINE on page 1172

The following statement illustrates the HOUR function:

Statements	Results
`select hour(endtime) from custonline;`	10 22 19 12 12 16 15 11 10

Statements	Results
	9
`select hour(current_time);`	14

See Also

- "Dates and Times in FedSQL" on page 58

Functions:

- "DAY Function" on page 420
- "MINUTE Function" on page 649
- "MONTH Function" on page 655
- "SECOND Function" on page 806
- "YEAR Function" on page 864

IBESSEL Function

Returns the value of the modified Bessel function.

Categories:	CAS Mathematical
Returned data type:	DOUBLE

Syntax

IBESSEL(*nu*, *x*, *kode*)

Arguments

nu

specifies a numeric constant, variable, or expression.

Range *nu* ≥ 0

Data type DOUBLE

x

specifies a numeric constant, variable, or expression.

Range $x \geq 0$

Data type DOUBLE

kode
is a numeric constant, variable, or expression that specifies a nonnegative whole number.

Data type DOUBLE

Details

The IBESSEL function returns the value of the modified Bessel function of order *nu* evaluated at *x* (Abramowitz, Stegun 1964; Amos, Daniel, Weston 1977). When *kode* equals 0, the Bessel function is returned. Otherwise, the value of the following function is returned:

$$\varepsilon^{-x} I_{nu}(x)$$

Example

The following statements illustrate the IBESSEL function:

Statements	Results
`select ibessel(2, 2, 0);`	`0.688948`
`select ibessel(2, 2, 1);`	`0.093239`

See Also

Functions:

- "JBESSEL Function" on page 605

IFNULL Function

Checks the value of the first expression and, if it is null or a SAS missing value, returns the second expression.

Categories: Special
CAS

Returned data type: Any data type, depending on the context

Syntax

IFNULL(*expression-1*, *expression-2*)

Arguments

expression
specifies any valid expression.

Data type All data types are valid.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The IFNULL function returns the first expression if it is not null. You can use the IFNULL function to replace a null or SAS missing value value with another value.

Comparisons

The NULLIF expression compares two expressions and returns a null value if the two expressions are equal. The IFNULL function checks the value of the first expression and if it is null, returns the second expression; otherwise, returns the first expression.

Example

Table: WORLDCITYCOORDS on page 1182

In the table WORLDTEMPS, the city for China in the last row contains a missing value.

Statements	Results

`select ifnull(AvgHigh, 0) as "AvgHigh: 0 indicates null/missing" from worldtemps;`	`AvgHigh: 0 indicates null/missing`
	`90`
	`70`
	`86`
	`90`

Statements	Results
	97 83 76 89 90 89 0 78

See Also

Expressions:

- “NULLIF Expression” on page 892

INDEX Function

Searches a character expression for a string of characters, and returns the position of the string's first character for the first occurrence of the string.

Categories:	CAS Character
Returned data type:	INTEGER

Syntax

INDEX(*target-expression*, *search-expression*)

Arguments

target-expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type	CHAR, NCHAR, NVARCHAR, VARCHAR
See	“<sql-expression> Expression” on page 894 “FedSQL Expressions” on page 46

search-expression
specifies any valid expression that evaluates or can be coerced to a character string that is used to search for in *target-expression*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip Enclose a literal string of characters in single quotation marks.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The INDEX function searches *target-expression*, from left to right, for the first occurrence of the string specified in *search-expression*, and returns the position in *target-expression* of the string's first character. If the string is not found in *target-expression*, INDEX returns a value of 0. If there are multiple occurrences of the string, INDEX returns only the position of the first occurrence.

Note: When either the *target-expression* or the *search-expression* has a length of 0, the INDEX function returns a value of 0. It should also be noted that an undeclared VARCHAR variable has a length of 0.

Comparisons

The VERIFY function returns the position of the first character in *target-expression* that does not contain *search-expression* where the INDEX function returns the position of the first occurrence of *search-expression* that is present in *target-expression*.

Example

The following statement illustrates the INDEX statement:

Statements	Results
`select index('aabb','ab');`	2

See Also

Functions:

- "INDEXC Function" on page 555
- "VERIFY Function" on page 855

INDEXC Function

Searches a character expression for specified characters and returns the position of the first occurrence of any of the characters.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

INDEXC(*target-expression*, *search-expression*[, …*search-expression*])

Arguments

target-expression

specifies any valid expression that evaluates or can be coerced to a character string that is searched.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

search-expression

specifies the characters to search for in *target-expression*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip Enclose a literal string of characters in single quotation marks.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The INDEXC function searches *target-expression*, from left to right, for the first occurrence of any character present in the search expressions and returns the position in *target-expression* of that character. If none of the characters in the search expressions are found in *target-expression*, INDEXC returns a value of 0.

Comparisons

The INDEXC function searches for the first occurrence of any individual character that is present within the search expression, whereas the INDEX function searches for the first occurrence of the search expression as a pattern.

Example

The following statement illustrates the INDEXC function:

Statements	Results
`select indexc('aabb','ab');`	`1`

See Also

Functions:

- "INDEX Function" on page 553

INPUTC Function

Enables you to specify a character informat at run time.

Categories: CAS
Special

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

INPUTC(*source*, *informat*[, *w*])

Arguments

source

specifies a character constant, variable, or expression to which you want to apply the informat.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

informat

is a character constant, variable, or expression that contains the character informat that you want to apply to *source*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

w

specifies any valid expression that evaluates to a numeric width to apply to the informat.

Interaction If you specify a width here, it overrides any width specification in the informat.

Data type DOUBLE

Details

If the INPUTC function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the length of the first argument.

Comparisons

The INPUTN function enables you to specify a numeric informat at run time.

Example

The following statements illustrate the INPUTC function:

Statements	Results
`select inputc('positive', '$upcase15.');`	`POSITIVE`
`select inputc('positive', '$upcase15.', 3);`	`POS`

See Also

Functions:

- “INPUTN Function” on page 558
- “PUT Function” on page 769

INPUTN Function

Enables you to specify a numeric informat at run time.

Categories:	CAS Special
Returned data type:	DOUBLE

Syntax

INPUTN(*source*, *informat*[, *w*[, *d*]])

Arguments

source

specifies a character constant, variable, or expression to which you want to apply the informat.

Data type CHAR

informat

is a character constant, variable, or expression that contains the numeric informat that you want to apply to *source*.

Data type CHAR

w

is a numeric constant, variable, or expression that specifies a width to apply to the informat.

Interaction If you specify a width here, it overrides any width specification in the informat.

Data type DOUBLE

d

is a numeric constant, variable, or expression that specifies the number of decimal places to use.

Interaction If you specify a number here, it overrides any decimal-place specification in the informat.

Data type DOUBLE

Comparisons

The INPUTC function enables you to specify a character informat at run time. Using the PUT function is faster because you specify the informat at compile time.

Example

The following statement illustrates the INPUTN function:

Statements	Results
`select inputn('20,000.00', 'comma10.2');`	`20000`

See Also

Functions:

- "INPUTC Function" on page 556
- "PUT Function" on page 769

INT Function

Returns the whole number, fuzzed to avoid unexpected floating-point results.

Categories: CAS
Truncation

Returned data type: DOUBLE

Syntax

INT(*expression*)

Arguments

expression
specifies any expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

In FedSQL, INT is a reserved word. To use the function, you must specify the function name as a delimited identifier.

The INT function returns the whole number portion of the argument (truncates the decimal portion). If the argument's value is within 1E-12 of a whole number, the function results in that whole number. If the value of *expression* is positive, the INT function has the same result as the FLOOR function. If the value of *expression* is negative, the INT function has the same result as the CEIL function.

Comparisons

Unlike the INTZ function, the INT function fuzzes the result. If the argument is within 1E-12 of a whole number, the INT function fuzzes the result to be equal to that whole number. The INTZ function does not fuzz the result. Therefore, with the INTZ function, you might get unexpected results.

Example

The following statements illustrate the INT function:

Statements	Results
`select "int"(2.6);`	`2`
`select "int"(-2.4);`	`-2`
`select "int"(1+1.3-11);`	`-8`
`select "int"(-1.6);`	`-1`

See Also

Functions:

- "CEIL Function" on page 358
- "FLOOR Function" on page 520
- "INTZ Function" on page 598

- "MOD Function" on page 651
- "MODZ Function" on page 653
- "ROUND Function" on page 786

INTCINDEX Function

Returns the cycle index when a date, time, or timestamp interval and value are specified.

Categories:	CAS Date and Time
Restriction:	FedSQL does not support custom date or time intervals.
Returned data type:	DOUBLE

Syntax

INTCINDEX({*interval*[*multiple*][.*shift-index*]}, *date-time-value*)

Arguments

interval

specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Note The possible values of *interval* are listed in "Intervals Used with Date and Time Functions" in *SAS Language Reference: Concepts*.

Tip *Interval* can appear in uppercase or lowercase.

Example YEAR specifies year-based intervals.

multiple

specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR2 specifies a two-year, or biennial, interval type.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

date-time-value
specifies a date, time, or timestamp value that represents a time period of a specified interval.

Data type DOUBLE

Details

The Basics

The INTCINDEX function returns the index of the seasonal cycle when you specify an interval and a DATE, TIME, or TIMESTAMP value. For example, if the interval is MONTH, each observation in the data corresponds to a particular month. Monthly data is considered to be periodic for a one-year period. A year contains 12 months, so the number of intervals (months) in a seasonal cycle (year) is 12. WEEK is the seasonal cycle for an interval that is equal to DAY. This example returns a value of 36 because September 1, 2013, is the sixth day of the 35th week of the year.

```
select intcindex('day', date'2013-09-01');
```

For more information about working with date and time intervals, see "Date and Time Intervals" in *SAS Functions and CALL Routines: Reference*.

The INTCINDEX function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For a list of these intervals, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Language Reference: Concepts*.

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval[*multiple*][.*shift-index*]

For more information, see "Arguments" on page 561.

Comparisons

The INTCINDEX function returns the cycle index, whereas the INTINDEX function returns the seasonal index.

In this example, the INTCINDEX function returns the week of the year.

```
select intcindex('day', date'2013-04-04');
```

In this example, the INTINDEX function returns the day of the week.

```
select intindex('day', date'2013-04-04');
```

In this example, the INTCINDEX function returns the hour of the day.

```
select intcindex('minute', timestamp'2012-09-01 00:00:00');
```

In this example, the INTINDEX function returns the minute of the hour.

```
select intindex('minute', timestamp'2012-09-01 00:00:00');
```

In the example `intseas(intcycle('interval'));`, the INTSEAS function returns the maximum number that could be returned by `intcindex('interval', date);`.

Example

The following statements illustrate the INTCINDEX function:

Statements	Results
`select intcindex('dtqtr', timestamp'2013-05-23 05:03:01');`	1
`select intcindex('tenday', date'2013-12-13');`	1
`select intcindex('minute', time'23:13:02');`	24
`select intcindex('semimonth', timestamp'2013-05-05 10:54:03');`	1

See Also

Functions:

- “INTCYCLE Function” on page 571
- “INTINDEX Function” on page 578
- “INTSEAS Function” on page 590

INTCK Function

Returns the number of interval boundaries of a given kind that lie between two SAS dates, times, or timestamp values encoded as DOUBLE.

Categories:	CAS Date and Time
Restriction:	FedSQL does not support custom date or time intervals.
Returned data type:	DOUBLE

Syntax

INTCK({*interval*[*multiple*][.*shift-index*]}, *start-date*, *end-date*[, '*method*'])

INTCK(*start-date*, *end-date*[, '*method*'])

Arguments

interval

specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Note The possible values of *interval* are listed in “Intervals Used with Date and Time Functions” in *SAS Language Reference: Concepts*.

Tip *Interval* can appear in uppercase or lowercase.

Example YEAR specifies year-based intervals.

multiple

specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR2 specifies a two-year, or biennial, interval type.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

start-date

specifies an expression that represents the starting SAS date, time, or timestamp value.

Data type DOUBLE

end-date

specifies an expression that represents the ending SAS date, time, or timestamp value.

Data type DOUBLE

'*method*'

specifies that intervals are counted using either a discrete or a continuous method.

You must enclose *method* in quotation marks. *Method* can be one of these values:

CONTINUOUS

specifies that continuous time is measured. The interval is shifted based on the starting date.

For example, the distance in months between January 15, 2013, and February 15, 2013, is one month.

Alias C or CONT

DISCRETE
specifies that discrete time is measured. The discrete method counts interval boundaries (for example, end of month).

The default discrete method is useful to sort time series observations into bins for processing. For example, daily data can be accumulated to monthly data for processing as a monthly series.

For the DISCRETE method, the distance in months between January 31, 2013, and February 1, 2013, is one month.

Alias D or DISC

Default DISCRETE

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval[*multiple*][.*shift-index*]

For more information, see “Arguments” on page 564.

Calendar Interval Calculations

All values within a discrete time interval are interpreted as being equivalent. This means that the dates of January 1, 2013 and January 15, 2013 are equivalent when you specify a monthly interval. Both of these dates represent the interval that begins on January 1, 2013 and ends on January 31, 2013. You can use the date for the beginning of the interval (January 1, 2013) or the date for the end of the interval (January 31, 2013) to identify the interval. These dates represent all of the dates within the monthly interval.

In the following example, the *start-date* (Jan. 14, 2013) is equivalent to the first quarter of 2013.

```
select intck('qtr', date'2013-01-14', date'2013-09-02)';
```

The *end-date* (September 2, 2013) is equivalent to the third quarter of 2013. The interval count, that is, the number of times the beginning of an interval is reached in moving from the *start-date* to the *end-date* is 2.

The INTCK function using the default discrete method counts the number of times the beginning of an interval is reached in moving from the first date to the second. It does not count the number of complete intervals between two dates:

- The following example returns 0, because the two dates are within the same month.

```
select intck('month', date'2013-01-01', date'2013-01-31');
```

- The following example returns 1, because the two dates lie in different months that are one month apart.

```
select intck('month', date'2013-01-31', date'2013-02-01');
```

- The following example returns –1 because the first date is in a later discrete interval than the second date. (INTCK returns a negative value whenever the first date is later than the second date and the two dates are not in the same discrete interval.)

```
select intck('month', date'2013-02-01', date'2013-01-31');
```

Using the discrete method, WEEK intervals are determined by the number of Sundays, the default first day of the week, that occur between the *start-date* and the *end-date*, and not by how many seven-day periods fall between those dates. To count the number of seven-day periods between *start-date* and *end-date*, use the continuous method.

Both the *multiple* and the *shift-index* arguments are optional and default to 1. For example, YEAR, YEAR1, YEAR.1, and YEAR1.1 are all equivalent ways of specifying ordinary calendar years.

For more information about working with date and time intervals, see "Date and Time Intervals" in *SAS Functions and CALL Routines: Reference*.

Intervals by Category

Table 4.2 *Intervals Used with Date and Time Functions*

Category	Interval	Definition	Default Starting Point	Shift Period	Example	Description
Date	DAY	Daily intervals	Each day	Days	DAY3	Three-day intervals starting on Sunday
	WEEK	Weekly intervals of seven days	Each Sunday	Days (1=Sunday … 7=Saturday)	WEEK.7	Weekly with Saturday as the first day of the week
	WEEKDAY <*days*W>	Daily intervals with Friday-Saturday-Sunday	Each day	Days	WEEKDAY1W	Six-day week with Sunday as a weekend day

Category	Interval	Definition	Default Starting Point	Shift Period	Example	Description
		counted as the same day (five-day work week with a Saturday-Sunday weekend). *days* identifies the weekend days by number (1=Sunday ... 7=Saturday). By default, *days*=17.			WEEKDAY35W	Five-day week with Tuesday and Thursday as weekend days (W indicates that day 3 and day 5 are weekend days)
	TENDAY	Ten-day intervals (a U.S. automobile industry convention)	First, 11th, and 21st of each month	Ten-day periods	TENDAY4.2	Four ten-day periods starting at the second TENDAY period
	SEMIMONTH	Half-month intervals	First and 16th of each month	Semi-monthly periods	SEMIMONTH2.2	Intervals from the 16th of one month through the 15th of the next month
	MONTH	Monthly intervals	First of each month	Months	MONTH2.2	February-March, April-May, June-July, August-September, October-November, and December-January of the following year
	QTR	Quarterly (three-	January 1	Months	QTR3.2	Three-month

Category	Interval	Definition	Default Starting Point	Shift Period	Example	Description
		month) intervals	April 1			intervals starting on April 1, July 1, October 1, and January 1
			July 1			
			October 1			
	SEMIYEAR	Semiannual (six-month) intervals	January 1	Months	SEMIYEAR.3	Six-month intervals, March-August, and September-February
			July 1			
	YEAR	Yearly intervals	January 1	Months		
Datetime	Add DT to any of the date intervals	Interval that corresponds to the associated date interval	Midnight of January 1, 1960		DTMONTH	
					DTWEEKDAY	
Time	SECOND	Second intervals	Start of the day (midnight)	Seconds		
	MINUTE	Minute intervals	Start of the day (midnight)	Minutes		
	HOUR	Hourly intervals	Start of the day (midnight)	Hours		

Retail Calendar Intervals

The retail industry often accounts for its data by dividing the yearly calendar into four 13-week periods, based on one of the following formats: 4-4-5, 4-5-4, or 5-4-4. The first, second, and third numbers specify the number of weeks in the first, second, and third month of each period, respectively. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Language Reference: Concepts*.

Example

The following statements illustrate the INTCK function:

Statements	Results
`select intck('qtr', date'2013-01-10', date'2013-07-01');`	2
`select intck('year', date'2012-12-31', date'2013-01-01');`	1
`select intck('year', date'2013-01-01', date'2013-12-31');`	0
`select intck('semiyear', date'2010-01-01', date'2013-01-01');`	6
`select intck('week2.2', date'2013-01-07', date'2013-04-01');`	6
`select intck('weekday7w', date'2013-01-01', date'2013-02-01');`	27
`select intck('year', date'2003-09-01', date'2013-09-01');`	10

In the second example, INTCK returns a value of 1 even though only one day has elapsed. This result is returned because the interval from December 31, 2012, to January 1, 2013, contains the starting point for the YEAR interval. However, in the third example, a value of 0 is returned even though 364 days have elapsed. This result occurs because the period between January 1, 2013, and December 31, 2013, does not contain the starting point for the interval.

In the fourth example, SAS returns a value of 6 because the period of January 1, 2010, through January 1, 2013, contains six semiyearly intervals. (Note that if the ending date were December 31, 2012, SAS would count five intervals.) In the fifth example, SAS returns a value of 6 because there are six two-week intervals beginning on a first Monday during the period of January 7, 2013, through April 1, 2013. In the sixth example, SAS returns the value 27. That indicates that beginning with January 1, 2013, and counting only Saturdays as weekend days through February 1, 2013, the period contains 27 weekdays. The last example specifies the number of years that have elapsed between the specified dates (10).

See Also

Other References:

- “Dates and Times in FedSQL” on page 58

Functions:

- “INTNX Function” on page 582

INTCYCLE Function

Returns the date, time, or datetime interval at the next higher seasonal cycle when a date, time, or datetime interval is specified.

Categories:	CAS Date and Time
Restriction:	FedSQL does not support custom date or time intervals.
Returned data type:	CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

INTCYCLE({*interval*/[*multiple*][.*shift-index*]}[, *seasonality*])

Arguments

interval

specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Note The possible values of *interval* are listed in "Intervals Used with Date and Time Functions" in *SAS Language Reference: Concepts*.

Tip *Interval* can appear in uppercase or lowercase.

Example YEAR specifies year-based intervals.

multiple

specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR2 specifies a two-year, or biennial, interval type.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but

YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

seasonality

specifies a numeric value.

This argument enables you to have more flexibility in working with dates and time cycles. You can specify whether you want a 52-week or a 53-week seasonality in a year.

Default 52

Data type DOUBLE, CHAR, NCHAR, NVARCHAR, VARCHAR

Example The *seasonality* argument in the following example

```
INTCYCLE('MONTH', 3);
```

causes the function call to return the value QTR. The function call

```
INTCYCLE('MONTH');
```

does not have a *seasonality* argument and returns the value YEAR.

Details

The Basics

The INTCYCLE function returns the interval of the seasonal cycle, depending on a date, time, or datetime interval. For example, `INTCYCLE('MONTH');` returns the value YEAR because the months from January through December constitute a yearly cycle. `INTCYCLE('DAY');` returns the value WEEK because the days from Sunday through Saturday constitute a weekly cycle.

For information about multipliers and shift indexes, see "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference*. For information about how intervals are calculated, see "Commonly Used Time Intervals" in *SAS Functions and CALL Routines: Reference*.

For more information about working with date and time intervals, see "Date and Time Intervals" in *SAS Functions and CALL Routines: Reference*..

The INTCYCLE function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Functions and CALL Routines: Reference*.

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval[*multiple*][.*shift-index*]

For more information, see "Arguments" on page 571.

Seasonality

Seasonality is a time series concept that measures cyclical variations at different intervals during the year. In specifying seasonality, the time of year is the most common source of the variations. For example, sales of home heating oil are regularly greater in winter than during other times of the year. Often, certain days of the week cause regular fluctuations in daily time series, such as increased spending on leisure activities during weekends. The INTCYCLE function uses the concept of seasonality and returns the date, time, or datetime interval at the next higher seasonal cycle when a date, time, or datetime interval is specified. For more information about seasonality and using the forecasting methods in PROC FORECAST, see the *SAS/ETS User's Guide*.

Example

The following statements illustrate the INTCYCLE function:

Statements	Results
`select intcycle('year');`	YEAR
`select intcycle('qtr');`	YEAR
`select intcycle('month', 3);`	QTR
`select intcycle('month');`	YEAR
`select intcycle('weekday');`	WEEK
`select intcycle('weekday', 5);`	WEEK

Statements	Results
`select intcycle('day');`	`WEEK`
`select intcycle('day', 10);`	`TENDAY`
`intcycle('day', 10);`	`DTMINUTE`

INTFIT Function

Returns a time interval that is aligned between two dates.

Categories: CAS
Date and Time

Restriction: FedSQL does not support custom date or time intervals.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

INTFIT(*expresssion-1*, *expression-2*, '*type*')

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string and that represents a SAS date or datetime value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

'*type*'
specifies whether the arguments are SAS date values, datetime values, or a row.

The following values for *type* are valid:

d specifies that *expression-1* and *expression-2* are date values.

dt specifies that *expression-1* and *expression-2* are datetime values.

obs specifies that *expression-1* and *expression-2* are rows.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The INTFIT function returns the most likely time interval based on two dates, datetime values, or rows that have been aligned within an interval. INTFIT assumes that the alignment value is SAME, which specifies that the date is aligned to the same calendar date with the corresponding interval increment. For more information about the *alignment* argument, see "INTNX Function" in *SAS Functions and CALL Routines: Reference*.

If the arguments that are used with INTFIT are rows, you can determine the cycle of an occurrence by using row numbers. In the following example, the first two arguments of INTFIT are row numbers, and the *type* argument is `obs`. If Jason used the gym the first time and the 25th time that a researcher recorded data, you could determine the interval by using the following statement: `interval=intfit(1, 25, 'obs');`. In this case, the value of interval is OBS24.2.

For information about time series, see the *SAS/ETS 9.3 User's Guide*.

The INTFIT function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Language Reference: Concepts*.

Example

The following statement illustrates the INTFIT function. This example shows the interval aligned between two dates.

Statements	Results
`select intfit(date'2013-08-01', date'2013-09-01', 'd');`	`month`

See Also

Functions:

- "INTCK Function" on page 564
- "INTNX Function" on page 582

INTGET Function

Returns a time interval based on three date or datetime values.

Categories: CAS
Date and Time

Restriction: FedSQL does not support custom date or time intervals.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

INTGET(*date-1*, *date-2*, *date-3*)

Argument

date

specifies any valid expression that evaluates to a SAS date or datetime value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

INTGET Function Intervals

The INTGET function returns a time interval based on three date or datetime values. The function first determines all possible intervals between the first two dates, and then determines all possible intervals between the second and third dates. If the intervals are the same, INTGET returns that interval. If the intervals for the first and second dates differ, and the intervals for the second and third dates differ, INTGET compares the intervals. If one interval is a multiple of the other, then INTGET returns the smaller of the two intervals. Otherwise, INTGET returns a missing value. INTGET works best with dates generated by the INTNX function whose alignment value is BEGIN.

In the following example, INTGET returns the interval DAY2:

```
select intget(date'2000-03-01', date'2000-03-03', date'2000-03-09');
```

The interval between the first and second dates is DAY2, because the number of days between March 1, 2000, and March 3, 2000, is two. The interval between the second and third dates is DAY6, because the number of days between March 3, 2000, and March 9, 2000, is six. DAY6 is a multiple of DAY2. INTGET returns the smaller of the two intervals.

In the following example, INTGET returns the interval MONTH4:

```
select intget(date'2000-01-01', date'2000-05-01', date'2001-05-01');
```

The interval between the first two dates is MONTH4, because the number of months between January 1, 2000, and May 1, 2000, is four. The interval between the second and third dates is YEAR. INTGET determines that YEAR is a multiple of MONTH4 (there are three MONTH4 intervals in YEAR), and returns the smaller of the two intervals.

In the following example, INTGET returns a missing value:

```
select intget(date'2006-01-01', date'2006-04-01', date'2006-12-01');
```

The interval between the first two dates is MONTH3, and the interval between the second and third dates is MONTH8. INTGET determines that MONTH8 is not a multiple of MONTH3, and returns a missing value.

The intervals that are returned are valid SAS intervals, including multiples of the intervals and shift intervals. Valid SAS intervals are listed in "Intervals Used with Date and Time Functions" in *SAS Language Reference: Concepts* .

Note: If INTGET cannot determine a matching interval, then the function returns a missing value. No message is written to the SAS log.

Retail Calendar Intervals

The INTGET function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Language Reference: Concepts*.

Example

The following statements illustrate the INTGET function:

Statements	Results
`select intget(date'2013-01-01', date'2014-01-01', date'2014-05-01');`	MONTH4
`select intget(date'2012-02-29', date'2014-02-28', date'2016-02-29');`	YEAR2.2
`select intget(date'2013-02-01', date'2013-02-16', date'2013-03-01');`	SEMIMONTH
`select intget(date'2013-01-02', date'2014-02-02', date'2015-03-02');`	MONTH13.13
`select intget(date'2013-02-10', date'2013-02-19', date'2013-02-28');`	DAY9.5
`select intget(timestamp'2014-04-01 01:03:00.0000', timestamp'2014-04-01 01:04:00.0000', timestamp'2014-04-01 01:05:00.0000');`	MINUTE

See Also

Functions:

- "INTFIT Function" on page 574
- "INTNX Function" on page 582

INTINDEX Function

Returns the seasonal index when a date, time, or timestamp interval and value are specified.

Categories:	CAS Date and Time
Restriction:	FedSQL does not support custom date or time intervals.
Returned data type:	DOUBLE

Syntax

INTINDEX({*interval*[*multiple*][.*shift-index*]}, *date-value*[, *seasonality*])

Arguments

interval

specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Note The possible values of *interval* are listed in "Intervals Used with Date and Time Functions" in *SAS Language Reference: Concepts*.

Tip *Interval* can appear in uppercase or lowercase.

Example YEAR specifies year-based intervals.

multiple

specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR2 specifies a two-year, or biennial, interval type.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

date-value

specifies a date, time, or timestamp value that represents a time period of the given interval.

Data type DOUBLE

seasonality

specifies a number or a cycle.

This argument enables you to have more flexibility in working with dates and time cycles. You can specify whether you want a 52-week or a 53-week seasonality in a year.

Data type DOUBLE, CHAR, NCHAR, NVARCHAR, VARCHAR

Example In this example, the following functions produce the same result.

```
INTINDEX('MONTH', sasdate, 3);
INTINDEX('MONTH', sasdate, 'QTR');
```

Seasonality in the first example is a number (the number of months), and in the second example *seasonality* is a cycle (QTR).

Details

The Basics

The INTINDEX function returns the seasonal index when you supply an interval and an appropriate date, time, or timestamp value. The seasonal index is a number that

represents the position of the date, time, or timestamp value in the seasonal cycle of the specified interval. This example returns a value of 12 because there are 12 months in a yearly cycle and December is the 12th month of the year.

```
select intindex('month', date'2012-12-01');
```

In the following examples, INTINDEX returns the same value (1) because both statements have dates that occur in the first quarter of the year 2013.

```
select intindex('qtr', date'2013-01-01');
select intindex('qtr', date'2013-03-01');
```

The following example returns a value of 6 because daily data is weekly periodic and December 7, 2012, is a Friday, the sixth day of the week.

```
select intindex('day', date'2012-12-07');
```

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval[*multiple*][.*shift-index*]

For more information, see “Arguments” on page 578.

How *Interval* and *Date-Time-Value* Are Related

```
sasdate=to_double(date'2012-12-07');
x=intindex('day', sasdate);
put x;
```

To correctly identify the seasonal index, the interval should agree with the date, time, or timestamp value. For example, `intindex('month', date'2012-12-01');` returns a value of 12 because there are 12 months in a yearly interval and December is the 12th month of the year. The MONTH interval requires a SAS date value. The following example, returns a value of 6 because there are seven days in a weekly interval and December 7, 2012, is a Friday, the sixth day of the week.

```
select intget('day', date'2012-12-07');
```

The DAY interval requires a SAS date value.

This example returns a missing value because the QTR interval expects the date to be a SAS date value rather than a TIMESTAMP value.

```
select intindex('qtr',timestamp'2013-01-01 00:00:00');
```

This example returns a value of 12. The DTMONTH interval requires a TIMESTAMP value.

```
select intindex('dtmonth',timestamp'2013-12-01 00:00:00');
```

For more information about working with date and time intervals, see “Date and Time Intervals” in *SAS Functions and CALL Routines: Reference*.

Retail Calendar Intervals

The INTINDEX function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Functions and CALL Routines: Reference*.

Seasonality

Seasonality is a time series concept that measures cyclical variations at different intervals during the year. In specifying seasonality, the time of year is the most common source of the variations. For example, sales of home heating oil are regularly greater in winter than during other times of the year. Often, certain days of the week cause regular fluctuations in daily time series, such as increased spending on leisure activities during weekends. The INTINDEX function uses the concept of seasonality and returns the seasonal index when a date, time, or timestamp interval and value are specified. For more information about seasonality and using the forecasting methods in PROC FORECAST, see the *SAS/ETS User's Guide*.

Comparisons

The INTINDEX function returns the seasonal index whereas the INTCINDEX function returns the cycle index.

In the following example, the INTINDEX function returns 5 because April 4, 2013 is on a Thursday, the fifth day of the week.

```
select intindex('day', date'2013-04-04');
```

Using the same date, the INTCINDEX function returns 14 because April 4, 2013 is the 14th week of the year.

```
select intcindex('day', date'2013-04-04');
```

In this example, the INTINDEX function returns the minute of the hour.

```
select intindex('minute', timestamp'2012-09-01 06:05:04');
```

Using the same date and time, the INTCINDEX function returns the hour of the day.

```
select intcindex('minute', timestamp'2012-09-01 06:05:04');
```

In the example `intseas('interval');`, INTSEAS returns the maximum number that could be returned by `intindex('interval', date);`.

Example

The following statements illustrate the INTINDEX function:

Statements	Results
`select intindex('qtr', date'2013-08-14');`	3

Statements	Results
`select intindex('dtqtr', timestamp'2013-12-23 15:09:19');`	4
`select intindex('hour', time'09:05:15');`	10
`select intindex('month', date '2013-02-26');`	2
`select intindex('dtmonth', timestamp'2013-05-28 05:15:00');`	5
`select intindex('week', date '2013-09-09');`	37
`select intindex('tenday', date '2013-04-16');`	11

See Also

Functions:

- “INTCINDEX Function” on page 561
- “INTCYCLE Function” on page 571
- “INTSEAS Function” on page 590

Other References:

- *SAS/ETS User's Guide*

INTNX Function

Increments a SAS date, time, or datetime value encoded as a DOUBLE, and returns a SAS date, time, or datetime value encoded as a DOUBLE.

Categories:	CAS Date and Time
Restriction:	FedSQL does not support custom date or time intervals.
Returned data type:	DOUBLE

Syntax

INTNX({*interval*[*multiple*][.*shift-index*]}, *start-from*, *increment*[, '*alignment*'])

INTNX(*start-from*, *increment*[, '*alignment*'])

Arguments

interval

specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Note The possible values of *interval* are listed in "Intervals Used with Date and Time Functions" in *SAS Language Reference: Concepts*.

Tip *Interval* can appear in uppercase or lowercase.

Example YEAR specifies year-based intervals.

multiple

specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR2 specifies a two-year, or biennial, interval type.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

start-from
specifies an expression that represents a SAS date, time, or datetime value encoded as a DOUBLE and that identifies a starting point.

Data type DOUBLE

increment
specifies a negative, positive, or zero whole number that represents the number of date, time, or datetime intervals. *Increment* is the number of intervals to shift the value of *start-from*.

Data type DOUBLE

'*alignment*'
controls the position of SAS dates within the interval. You must enclose *alignment* in quotation marks. *Alignment* can be one of these values:

BEGINNING
specifies that the returned date or datetime value is aligned to the beginning of the interval.

Alias B

MIDDLE
specifies that the returned date or datetime value is aligned to the midpoint of the interval, which is the average of the beginning and ending alignment values.

Alias M

END
specifies that the returned date or datetime value is aligned to the end of the interval.

Alias E

SAME
specifies that the date that is returned has the same alignment as the input date.

Aliases S

SAMEDAY

See “SAME Alignment” on page 585

Default BEGINNING

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See “Aligning SAS Date Output within Its Intervals” on page 585

Details

The Basics

The INTNX function increments a date, time, or datetime value by intervals such as DAY, WEEK, QTR, and MINUTE, or a custom interval that you define. The increment is based on a starting date, time, or datetime value, and on the number of time intervals that you specify.

The INTNX function returns the SAS date value for the beginning date, time, or datetime value of the interval that you specify in the *start-from* argument. (To convert the date value to a calendar date, use any valid DS2 date format, such as the DATE9. format.) The following example shows how to determine the date of the start of the week that is six weeks from the week of October 17, 2011.

```
select put(intnx('week', date'2011-10-17', 6), date9.);
```

INTNX returns the value 27NOV2011.

For more information about working with date and time intervals, see "Date and Time Intervals" in *SAS Functions and CALL Routines: Reference*.

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval[*multiple*][.*shift-index*]

For more information, see "Arguments" on page 583.

Aligning SAS Date Output within Its Intervals

SAS date values are typically aligned with the beginning of the time interval that is specified with the *interval* argument.

You can use the optional *alignment* argument to specify the alignment of the date that is returned. The values BEGINNING, MIDDLE, or END align the date to the beginning, middle, or end of the interval, respectively.

SAME Alignment

If you use the SAME value of the *alignment* argument, then INTNX returns the same calendar date after computing the interval increment that you specified. The same calendar date is aligned based on the interval's shift period, not the interval. To view the valid shift periods, see "Intervals by Category" on page 567.

Most of the values of the shift period are equal to their corresponding intervals. The exceptions are the intervals WEEK, WEEKDAY, QTR, SEMIYEAR, YEAR, and their DT counterparts. WEEK and WEEKDAY intervals have a shift period of DAYS; and QTR, SEMIYEAR, and YEAR intervals have a shift period of MONTH. For example, when you use SAME alignment with YEAR, the result is same-day alignment based

on MONTH, the interval's shift period. The result is not aligned to the same day of the YEAR interval. If you specify a multiple interval, then the default shift interval is based on the interval, and not on the multiple interval.

When you use SAME alignment for QTR, SEMIYEAR, and YEAR intervals, the computed date is the same number of months from the beginning of the interval as the input date. The day of the month matches as closely as possible. Because not all months have the same number of days, it is not always possible to match the day of the month.

For more information about shift periods, see “Intervals by Category” on page 567.

Alignment Intervals

Use the SAME value of the *alignment* argument if you want to base the alignment of the computed date on the alignment of the input date.

```
/*** returns 22MAR2011 ***/
select put(intnx('week', date'2011-03-15', 1, 'same'), date9.);

/*** returns 15MAR2016 ***/
select put(intnx('year', date'2011-03-15', 5, 'same'), date9.);
```

Adjusting Dates

The INTNX function automatically adjusts for the date if the date in the interval that is incremented does not exist. Here is an example:

```
/*** returns 15AUG2011 ***/
select put(intnx('month', date'2011-03-15', 5, 'same'), date9.);

/*** returns 28FEB2014 ***/
select put(intnx('year', date'2012-02-29', 2, 'same'), date9.);

/*** returns 30SEP2011 ***/
select put(intnx('month', date'2011-08-31', 1, 'same'), date9.);

/*** returns 01MAR2012 (the 1st day of the 3rd month of the year) ***/
select put(intnx('year', date'2011-03-01', 1, 'same'), date9.);

/*** returns 29FEB2012 (the 60th day of the year) ***/
select put(intnx('year', date'2011-03-01', 1, 'same', 'day'), date9.);
```

In the following example, the INTNX function returns the value 01JAN2014, which is the beginning of the year two years from the starting date (29FEB2012).

```
select put(intnx('year', date'2012-02-29', 2), date9.);
```

In this example , the INTNX function returns the value 28FEB2014. In this case, the starting date begins in the year 2012, the year is two years later (2014), the month is the same (February), and the date is the 28th, because that is the closest date to the 29th in February 2014.

```
select put(intnx('year', date'2012-02-29', 2, 'same'), date9.);
```

Retail Calendar Intervals

The retail industry often accounts for its data by dividing the yearly calendar into four 13-week periods, based on one of the following formats: 4-4-5, 4-5-4, or 5-4-4. The first, second, and third numbers specify the number of weeks in the first, second, and third month of each period, respectively. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Language Reference: Concepts*.

Examples

Example 1: Using the INTNX Function

The following statements illustrate the INTNX function.

Statements	Results
`select intnx('year', date'2013-02-05', 3);` `select put(intnx('year', date'2013-02-05', 3), date7.);`	20454 01Jan16
`select intnx('month', date'2013-01-05', 0);` `select put(intnx('month', date'2013-01-05', 0), date7.);`	19359 01JAN13
`select intnx('semiyear', date'2013-01-01', 1);` `select put(intnx('semiyear', date'2013-01-01', 1), date7.);`	19540 01JUL13
`select intnx('month2', date'2012-08-01', -1);` `select put(intnx('month2', date'2012-08-01', -1), date7.);`	19114 01MAY12

`select intnx('semimonth2.2', date'2013-04-01', 4);`
`select put(intnx('semimonth2.2', date'2013-04-01',`
`4), date7.);`
	19555
	16JUL13

Example 2: Using the ALIGNMENT Argument

The following examples show the results of advancing a date by using the optional *alignment* argument.

Statements	Results

`select intnx('month', date'2013-01-01', 5, 'beginning');`
`select put(intnx('month', date'2013-01-01', 5, 'beginning'),`
` date7.);`
	19510
	01JUN13

`select intnx('month', date'2013-01-01', 5, 'middle');`
`select put(intnx('month', date'2013-01-01', 5, 'middle'),`
` date7.);`
	19524
	15JUN13

`select intnx('month', date'2013-01-01', 5, 'end');` `select put(intnx('month', date'2013-01-01', 5, 'end'),`	19539 30JUN13

Statements	Results
date7.);	
select intnx('month', date'2013-01-01', 5, 'sameday'); select put(intnx('month', date'2013-01-01', 5, 'sameday'), date7.);	19510 01JUN13
select intnx('month', date'2013-03-15', 5, 'same'); select put(intnx('month', date'2013-03-15', 5, 'same'), date7.);	19585 15AUG13

See Also

Functions:

- "INTCK Function" on page 564

Other References:

- "Dates and Times in FedSQL" on page 58

INTRR Function

Returns the internal rate of return as a decimal value.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

INTRR(*freq*, *c0, c1*[..., *cn*])

Arguments

freq

is numeric, the number of payments over a specified base period of time that is associated with the desired internal rate of return.

Range *freq* > 0

Data type DOUBLE

Tip The case *freq* = 0 is a flag to allow continuous compounding.

***c0, c1*[…, *cn*]**
are numeric, the optional cash payments.

Data type DOUBLE

Details

The INTRR function returns the internal rate of return over a specified base period of time for the set of cash payments *c0*, *c1*, ..., *cn*. The time intervals between any two consecutive payments are assumed to be equal. The argument *freq* > 0 describes the number of payments that occur over the specified base period of time. The number of notes issued from each instance is limited.

The internal rate of return is the interest rate such that the sequence of payments has a 0 net present value. (See the "NPV Function" on page 685.) It is given by the following equation.

$$r = \begin{cases} \frac{1}{x^{freq}} - 1 & freq > 0 \\ -log_{\varepsilon}(x) & freq = 0 \end{cases}$$

In this equation, *x* is the real root of the polynomial.

$$\sum_{i=0}^{n} c_i x^i = 0$$

In the case of multiple roots, one real root is returned and a warning is issued concerning the non-uniqueness of the returned internal rate of return. Depending on the value of payments, a root for the equation does not always exist. In that case, a missing value is returned.

Missing values in the payments are treated as 0 values. When *freq* > 0, the computed rate of return is the effective rate over the specified base period. To compute a quarterly internal rate of return (the base period is three months) with monthly payments, set *freq* to 3.

If *freq* is 0, continuous compounding is assumed and the base period is the time interval between two consecutive payments. The computed internal rate of return is the nominal rate of return over the base period. To compute with continuous compounding and monthly payments, set *freq* to 0. The computed internal rate of return is a monthly rate.

Comparisons

The IRR function is identical to INTRR, except for in the IRR function, the internal rate of return is a percentage.

Example

For an initial outlay of $400 and expected payments of $100, $200, and $300 over the following three years, the annual internal rate of return can be calculated as follows:

```
select intrr(1,-400,100,200,300);
```

```
0.19437709962747
```

SAS writes the following output to the log.

```
0.1943770996
```

See Also

Functions:

- "IRR Function" on page 603

INTSEAS Function

Returns the length of the seasonal cycle when a date, time, or datetime interval is specified.

Categories:	CAS Date and Time
Restriction:	FedSQL does not support custom date or time intervals.
Returned data type:	DOUBLE

Syntax

INTSEAS({*interval*[*multiple*][.*shift-index*]}[, *seasonality*])

Arguments

interval

specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Note The possible values of *interval* are listed in "Intervals Used with Date and Time Functions" in *SAS Language Reference: Concepts*.

Tip *Interval* can appear in uppercase or lowercase.

Example YEAR specifies year-based intervals.

multiple

specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR2 specifies a two-year, or biennial, interval type.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

seasonality

specifies a numeric value.

This argument enables you to have more flexibility in working with dates and time cycles. You can specify whether you want a 52-week or a 53-week seasonality in a year.

Default 52

Data type DOUBLE, CHAR, NCHAR, NVARCHAR, VARCHAR

Example The *seasonality* argument in the following example

```
INTSEAS('MONTH', 'qtr');
```

causes the function call to return the value 3. The function call

```
INTSEAS('MONTH');
```

does not have a *seasonality* argument and returns the value 12.

Details

The Basics

The INTSEAS function returns the number of intervals in a seasonal cycle. For example, when the interval for a time series is described as monthly, then many procedures use the option INTERVAL=MONTH. Each observation in the data then corresponds to a particular month. Monthly data is considered to be periodic for a one-year period. A year contains 12 months, so the number of intervals (months) in a seasonal cycle (year) is 12.

Quarterly data is also considered to be periodic for a one-year period. A year contains four quarters, so the number of intervals in a seasonal cycle is four.

The periodicity is not always one year. For example, INTERVAL=DAY is considered to have a period of one week. Because there are seven days in a week, the number of intervals in the seasonal cycle is seven.

For more information about working with date and time intervals, see "Date and Time Intervals" in *SAS Functions and CALL Routines: Reference*.

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval[*multiple*][.*shift-index*]

For more information, see "Arguments" on page 590.

Retail Calendar Intervals

The retail industry often accounts for its data by dividing the yearly calendar into four 13-week periods, based on one of the following formats: 4-4-5, 4-5-4, or 5-4-4. The first, second, and third numbers specify the number of weeks in the first, second, and third month of each period, respectively. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Language Reference: Concepts*.

Seasonality

Seasonality is a time series concept that measures cyclical variations at different intervals during the year. In specifying seasonality, the time of year is the most common source of the variations. For example, sales of home heating oil are regularly greater in winter than during other times of the year. Often, certain days of the week cause regular fluctuations in daily time series, such as increased spending on leisure activities during weekends. The INTSEAS function uses the concept of seasonality and returns the length of the seasonal cycle when a date, time, or datetime interval is specified. For more information about seasonality and forecasting, see the *SAS/ETS User's Guide*.

Example

The following statements illustrate the INTCYCLE function:

Statements	Results
`select intseas('year');`	1
`select intseas('semiyear');`	2
`select intseas('quarter');`	4
`select intseas('month');`	12
`select intseas('semimonth');`	24
`select intseas('tenday');`	36
`select intseas('week');`	52
`select intseas('weekday');`	5
`select intseas('hour');`	24
`select intseas('minute');`	60
`select intseas('month2.2');`	6
`select intseas('week2');`	26
`select intseas('month4.3');`	3
`select intseas('day1');`	7

See Also

Functions:

- "INTCYCLE Function" on page 571
- "INTINDEX Function" on page 578

Other References:

- *SAS/ETS User's Guide*

INTSHIFT Function

Returns the shift interval that corresponds to the base interval.

Categories:	CAS Date and Time
Restriction:	FedSQL does not support custom date or time intervals.
Returned data type:	CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

INTSHIFT(*interval*[*multiple*][.*shift-index*])

Arguments

interval

specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Note The possible values of *interval* are listed in "Intervals Used with Date and Time Functions" in *SAS Language Reference: Concepts*.

Tip *Interval* can appear in uppercase or lowercase.

Example YEAR specifies yearly intervals.

multiple

specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR2 consists of two-year, or biennial, periods.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

Details

The Basics

The INTSHIFT function returns the shift interval that corresponds to the base interval. For custom intervals, the value that is returned is the base custom interval name. INTSHIFT ignores multiples of the interval and interval shifts.

The INTSHIFT function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Language Reference: Concepts*.

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR.

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval[*multiple*][.*shift-index*]

For more information, see "Arguments" on page 594.

Example

The following statements illustrate the INTSHIFT function:

Statements	Results
`select intshift('year');`	`MONTH`
`select intshift('dtyear');`	`DTMONTH`
`select intshift('minute');`	`DTMINUTE`
`select intshift('weekdays');`	`WEEKDAY`
`select intshift('weekday5.4');`	`WEEKDAY`
`select intshift('qtr');`	`MONTH`
`select intshift('dttenday');`	`DTTENDAY`

INTTEST Function

Returns 1 if a time interval is valid, and returns 0 if a time interval is invalid.

Categories: CAS
Date and Time

Restriction: FedSQL does not support custom date or time intervals.

Returned data type: DOUBLE

Syntax

INTTEST(*interval*[*multiple*][.*shift-index*])

Arguments

interval

specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Note The possible values of *interval* are listed in "Intervals Used with Date and Time Functions" in *SAS Language Reference: Concepts*.

Tip *Interval* can appear in uppercase or lowercase.

Example YEAR specifies year-based intervals.

multiple

specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR2 specifies a two-year, or biennial, interval type.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

Details

The Basics

The INTTEST function checks for a valid interval name. This function is useful when checking for valid values of *multiple* and *shift-index*. For more information about multipliers and shift indexes, see “Multiunit Intervals” in *SAS Language Reference: Concepts*.

The INTTEST function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see “Retail Calendar Intervals: ISO 8601 Compliant” in *SAS Language Reference: Concepts*.

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval[*multiple*][.*shift-index*]

For more information, see “Arguments” on page 597.

Example

In the following examples, SAS returns a value of 1 if the *interval* argument is valid, and 0 if the interval argument is invalid.

Statements	Results
`select inttest('month');`	1
`select inttest('week6.13');`	1
`select inttest('tenday');`	1
`select inttest('twoweeks');`	0
`select inttest('hour2.2');`	1

INTZ Function

Returns the whole number portion of the argument, using zero fuzzing.

Categories:	CAS Truncation
Returned data type:	DOUBLE

Syntax

INTZ(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The following rules apply:

- If the value of the argument is an exact whole number, INTZ returns that whole number.
- If the argument is positive and not a whole number, INTZ returns the largest whole number that is less than the argument.
- If the argument is negative and not a whole number, INTZ returns the smallest whole number that is greater than the argument.

Comparisons

Unlike the INT function, the INTZ function uses zero fuzzing. If the argument is within 1E-12 of a whole number, the INT function fuzzes the result to be equal to that whole number. The INTZ function does not fuzz the result. Therefore, with the INTZ function, you might get unexpected results.

Example

The following statements illustrate the INTZ function:

Statements	Results
`select intz('2.1');`	`2`
`select intz(-2.4);`	`-2`
`select intz(1+1.e11);`	`1E11`
`select intz(-1.6);`	`-1`

See Also

Functions:

- “CEIL Function” on page 358
- “CEILZ Function” on page 359
- “FLOOR Function” on page 520
- “FLOORZ Function” on page 521
- “INT Function” on page 559
- “MOD Function” on page 651
- “MODZ Function” on page 653
- “ROUND Function” on page 786
- “ROUNDZ Function” on page 789

IPMT Function

Returns the interest payment for a given period for a constant payment loan or the periodic savings for a future balance.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

IPMT(*rate*, *period*, *number-of-periods*, *principal-amount*[, *future-amount*][, *type*])

Arguments

rate

specifies the interest rate per payment period.

Data type DOUBLE

period

specifies the payment period for which the interest payment is computed.

Requirement *Period* must be a positive whole number that is less than or equal to the value of *number-of-periods*.

Data type INTEGER

number-of-periods

specifies the number of payment periods.

Requirement *Number-of-periods* must be a positive whole number.

Data type INTEGER

principal-amount

specifies the principal amount of the loan.

Data type DOUBLE

Note Zero is assumed if a missing value is specified.

future-amount

specifies the future amount.

Data type DOUBLE

Notes *Future-amount* can be the outstanding balance of a loan after the specified number of payment periods, or the future balance of periodic savings.

Zero is assumed if *future-amount* is omitted or if a missing value is specified.

type

specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments.

Data type INTEGER

Note 0 is assumed if *type* is omitted or if a missing value is specified.

Example

The interest payment on the first periodic payment of an $8,000 loan, where the nominal annual interest rate is 10% and the end-of-period monthly payments are 36, is computed as follows:

```
select ipmt(0.1/12, 1, 36, 8000);
```

```
66.6666666666666
```

SAS writes the following output to the log.

```
66.66667
```

If the same loan has beginning-of-period payments, then the interest payment can be computed as follows:

- `select ipmt(0.1/12, 1, 36, 8000, 0, 1);`

  This computation returns a value of 0.

- `select ipmt(0.1, 3, 3, 8000);`

  This computation returns a value of 292.4471.

- `select ipmt(0.09/12, 359, 360, 125000, 0, 1);`

  This computation returns a value of 14.80757.

IQR Function

Returns the interquartile range.

Categories:	CAS Descriptive Statistics
Returned data type:	DOUBLE

Syntax

IQR(*expression* [, …*expression*])

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Data type	DOUBLE
See	“<sql-expression> Expression” on page 894 “FedSQL Expressions” on page 46

Details

If all arguments have null or missing values, the result is a null or missing value depending on whether you are in ANSI mode or SAS mode. For more information, see "How FedSQL Processes Nulls and SAS Missing Values" on page 20.

Otherwise, the result is the interquartile range of the non-null or nonmissing values. The formula for the interquartile range is the same as the one that is used in the Base SAS UNIVARIATE procedure. For more information, see *Base SAS Procedures Guide: Statistical Procedures*.

Example

The following statement illustrates the IQR function:

Statements	Results
`select iqr(2,4,1,3,999999);`	2

See Also

Functions:

- "MAD Function" on page 632
- "PCTL Function" on page 691

IRR Function

Returns the internal rate of return as a percentage.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

IRR(*freq*, *c1, c2* [..., *cn*])

Arguments

freq

is numeric, the number of payments over a specified base period of time that is associated with the desired internal rate of return.

Range *freq* > 0.

Data type DOUBLE

Tip The case *freq* = 0 is a flag to allow continuous compounding.

***c1, c2*[…, *cn*]**

are numeric, the optional cash payments.

Requirement At minimum, two cash payment values are required.

Data type DOUBLE

Details

The IRR function returns the internal rate of return over a specified base period of time for the set of cash payments *c1*, *c2*, …, *cn*. The time intervals between any two consecutive payments are assumed to be equal. The argument *freq* > 0 describes the number of payments that occur over the specified base period of time. The number of notes issued from each instance is limited.

Comparisons

The IRR function is identical to INTRR, except that in the IRR function, the internal rate of return is a percentage.

Example

For an initial outlay of $400 and expected payments of $100, $200, and $300 over the following three years, the annual internal rate of return as a percentage can be expressed by the following program:

```
select irr(1,-400,100,200,300);
```

The value that is returned is 19.437709963.

See Also

Functions:

- “INTRR Function” on page 588

JBESSEL Function

Returns the value of the Bessel function.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

JBESSEL(*nu*, *x*)

Arguments

nu
specifies a numeric constant, variable, or expression.

Range $nu \geq 0$

x
specifies a numeric constant, variable, or expression.

Range $x \geq 0$

Details

The JBESSEL function returns the value of the Bessel function of order *nu* evaluated at *x* (For more information, see Abramowitz and Stegun 1964; Amos, Daniel, and Weston 1977).

Example

The following statements illustrate the JBESSEL function:

Statements	Results
`select jbessel(2, 2);`	`0.352834`

See Also

Functions:

- "IBESSEL Function" on page 550

JULDATE Function

Returns the Julian date from a SAS date value.

Categories:	CAS Date and Time
Returned data type:	DOUBLE

Syntax

JULDATE(*date*)

Arguments

date

specifies any valid expression that represents a SAS date value.

Data type	DOUBLE
See	"<sql-expression> Expression" on page 894 "FedSQL Expressions" on page 46

Details

A SAS date value is a number that represents the number of days from January 1, 1960 to a specific date. The JULDATE function converts a SAS date value to a Julian date. If *date* falls within the 100-year span defined by the system option YEARCUTOFF=, the result has three, four or five digits: In a five-digit result, the first two digits represent the year, and the next three digits represent the day of the year (1 to 365, or 1 to 366 for leap years). As leading zeros are dropped from the result, the year portion of a Julian date can be omitted (for years ending in 00) or it can have only one digit (for years ending 01–09). Otherwise, the result has seven digits: the first four digits represent the year, and the next three digits represent the day of the year.

For years that end between 00–09, you can format the five-digit Julian date by using the Z5. format.

For more information about how FedSQL handles dates, see "Dates and Times in FedSQL" on page 58.

Comparisons

The function JULDATE7 is similar to JULDATE except that JULDATE7 always returns a four-digit year. Thus, JULDATE7 is year 2000 compliant because it eliminates the need to consider the implications of a two-digit year.

Example

The following statements illustrate the JULDATE function:

Statements	Results
`select juldate(mdy(12,31,2013));`	`7365`
`select put(juldate(mdy(12,31,2013)),z5.);`	`07365`
`select juldate(mdy(9,1,1999));`	`99244`
`select juldate(mdy(7,1,1886));`	`1886182`

See Also

Functions:

- "DATEJUL Function" on page 417
- "JULDATE7 Function" on page 607

JULDATE7 Function

Returns a seven-digit Julian date from a SAS date value.

Categories:	CAS Date and Time
Returned data type:	DOUBLE

Syntax

JULDATE7(*date*)

Arguments

date

specifies any valid expression that represents a SAS date value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

A SAS date value is a number that represents the number of days from January 1, 1960 to a specific date. The JULDATE7 function returns a seven-digit Julian date from a SAS date value. The first four digits represent the year, and the next three digits represent the day of the year.

For more information about how FedSQL handles dates, see "Dates and Times in FedSQL" on page 58.

Comparisons

The function JULDATE7 is similar to JULDATE except that JULDATE7 always returns a four-digit year. Thus, JULDATE7 is year 2000 compliant because it eliminates the need to consider the implications of a two-digit year.

Example

The following statements illustrate the JULDATE7 function:

Statements	Results
`select juldate7(mdy(12,31,2006));`	`2007365`
`select juldate7(mdy(12,31,2016));`	`2016366`

See Also

Functions:

- "JULDATE Function" on page 606

KURTOSIS Function

Returns the kurtosis of all values in an expression.

Categories:	Aggregate Descriptive Statistics CAS
Returned data type:	DOUBLE

Syntax

KURTOSIS([DISTINCT] *expression*)

Arguments

expression
specifies any valid SQL expression.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

DISTINCT
(optional) eliminates duplicate rows from the aggregate calculation.

Details

Kurtosis is primarily a measure of the heaviness of the tails of a distribution. Large values of kurtosis indicate that the distribution has heavy tails.

Null values and SAS missing values are ignored and are not included in the computation.

At least four nonnull or nonmissing arguments are required. Otherwise, the function returns a null value. If all nonnull or nonmissing arguments have equal values, the kurtosis is mathematically undefined and the KURTOSIS function returns a null value.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Example

Table: WORLDTEMPS on page 1184

The following statement illustrates the KURTOSIS function:

Statements	Results
`select kurtosis(AvgLow) from worldtemps;`	`-0.87431`

See Also

Functions:

- "STDDEV Function" on page 819

SELECT Statement Clauses:

- "SELECT Clause" on page 956
- "GROUP BY Clause" on page 968
- "HAVING Clause" on page 970

LARGEST Function

Returns the *k*th largest non-null or nonmissing value.

Categories:	CAS Descriptive Statistics
Returned data type:	DOUBLE

Syntax

LARGEST(*k*, *expression* [, …*expression*])

Arguments

k

specifies any valid expression that evaluates to a numeric value that represents the largest value to return. For example, if *k* is 2, the LARGEST function returns the second largest value from the list of expressions.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

expression

specifies any valid expression that evaluates to a numeric value and that is to be searched.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

If *k* is null or missing, less than zero, or greater than the number of values, the result is a null or missing value. Otherwise, if *k* is greater than the number of non-null or nonmissing values, the result is a null or missing value.

Example

The following statements illustrate the LARGEST function:

Statements	Results
`select largest(1, 456, 789, .Q, 123);`	`789`
`select largest(2, 456, 789, .Q, 123);`	`456`
`select largest(3, 456, 789, .Q, 123);`	`123`
`select largest(4, 456, 789, .Q, 123);`	`.`

See Also

Functions:

- "ORDINAL Function" on page 690
- "PCTL Function" on page 691
- "SMALLEST Function" on page 812

LCM Function

Returns the least common multiple for a set of whole numbers.

Categories:	CAS Mathematical
Returned data type:	DOUBLE

Syntax

LCM(*expression*-1, *expression*-2 [, …*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a whole number.

Requirement	At least two arguments are required.
Data type	DOUBLE
Note	If any expression evaluates to zero, an error occurs and a missing value is returned.
See	"<sql-expression> Expression" on page 894 "FedSQL Expressions" on page 46

Details

The least common multiple is the smallest number that two or more numbers divide into evenly.

Example

The following statements illustrate the LCM function:

Statements	Results
`select lcm(1,5,3,0);`	.
`select lcm(25,70,85,130);`	77350
`select lcm(33,78);`	858

See Also

Functions:

- "GCD Function" on page 536

LCOMB Function

Computes the logarithm of the COMB function, which is the logarithm of the number of combinations of *n* objects taken *r* at a time.

Categories: CAS
Combinatorial

Returned data type: DOUBLE

Syntax

LCOMB(*n*, *r*)

Arguments

n
is a nonnegative whole number that represents the total number of elements from which the sample is chosen.

r
is a nonnegative whole number that represents the number of chosen elements.

Restriction $r \leq n$

Comparisons

The LCOMB function computes the logarithm of the COMB function.

Example

The following statements illustrate the LCOMB function:

Statements	Results
`select lcomb(5000, 500);`	`1621.441`
`select lcomb(100, 10);`	`30.48232`

See Also

Functions:

- "COMB Function" on page 367

LEFT Function

Left aligns a character expression.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

LEFT(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

LEFT returns a character string with leading blanks moved to the end of the value.

Example

The following statements illustrate the LEFT function:

Statements	Results
	`----+----1----+---2--`
`select left('  END-OF-YEAR');`	`END-OF-YEAR`

See Also

Functions:

- "COMPRESS Function" on page 375

LFACT Function

Computes the logarithm of the FACT (factorial) function.

Categories: CAS
Combinatorial

Returned data type: DOUBLE

Syntax

LFACT(*n*)

Arguments

n
is a whole number that represents the total number of elements from which the sample is chosen.

Details

The LFACT function computes the logarithm of the FACT function.

Example

The following statements illustrate the LFACT function:

Statements	Results
`select lfact(5000);`	`37591.14`
`select lfact(100);`	`363.7394`

See Also

Functions:

- "FACT Function" on page 442

LGAMMA Function

Returns the natural logarithm of the Gamma function.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

LGAMMA(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement Must be a positive number.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Example

The following statements illustrate the LGAMMA function:

Statements	Results
`select lgamma(2);`	`0`
`select lgamma(1.5);`	`-0.120782238`

LOG Function

Returns the natural logarithm (base e) of a numeric value expression.

Categories: Mathematical
CAS

Returned data type: DOUBLE

Syntax

LOG(*expression*)

Arguments

expression
specifies any valid SQL expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Example

The following statements illustrate the LOG function:

Statements	Results
`select log(1.0);`	`0`
`select log(10.0);`	`2.302585`

See Also

Functions:

- "E Function" on page 436
- "LOG10 Function" on page 620
- "LOG2 Function" on page 619

LOG1PX Function

Returns the log of 1 plus the argument.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

LOG1PX(*x*)

Arguments

x

specifies a numeric variable, constant, or expression.

Data type DOUBLE

Details

The LOG1PX function computes the log of 1 plus the argument. The LOG1PX function is mathematically defined by the following equation, where –1 < x:

$$LOG1PX(x) = log(1 + x)$$

When x is close to 0, `LOG1PX(x)` can be more accurate than `LOG(1+x)`.

Example

The following statements illustrate the LOG1PX function. The first example computes the log of 1 plus the value 0.5. The second computes the value of X by using the LOG1PX function and the value of Y by using the LOG function. The PUT function applies the HEX16. format to show the full precision of which FedSQL is capable.

Statements	Results
`select log1px(0.5);`	`0.405465`
`select put(log1px(1.e-5), hex16.) as x,` `put(log(1+1.e-5), hex16.) as y;`	`x=3EE4F8AEA9AE7317` `y=3EE4F8AEA9AF0A25`

See Also

Functions:

- "LOG Function" on page 617

LOG2 Function

Returns the base-2 logarithm of a numeric value expression.

Categories: Mathematical
CAS

Returned data type: DOUBLE

Syntax

LOG2(*expression*)

Arguments

expression
specifies any valid SQL expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Example

The following statements illustrate the LOG2 function:

Statements	Results
`select log2(8.0);`	3
`select log2(4);`	2

See Also

Functions:

- "LOG Function" on page 617
- "LOG10 Function" on page 620

LOG10 Function

Returns the base-10 logarithm of a numeric value expression.

Categories: Mathematical
CAS

Returned data type: DOUBLE

Syntax

LOG10(*expression*)

Arguments

expression
specifies any valid SQL expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Example

The following statements illustrate the LOG10 function:

Statements	Results
`select log10(1.0);`	`0`
`select log10(10.0);`	`1`
`select log10(100.0);`	`2`

See Also

Functions:

- "LOG Function" on page 617
- "LOG2 Function" on page 619

LOGBETA Function

Returns the logarithm of the beta function.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

LOGBETA(*a*, *b*)

Arguments

a

is the first shape parameter, where *a*>0.

Data type DOUBLE

b

is the second shape parameter, where *b*>0.

Data type DOUBLE

Details

The LOGBETA function is mathematically given by the equation

$$log(\beta(a,b)) = log(\Gamma(a)) + log(\Gamma(b)) - log(\Gamma(a+b))$$

In the equation, $\Gamma(.)$ is the gamma function.

If the expression cannot be computed, LOGBETA returns a missing value.

Example

The following statement illustrates the LOGBETA function:

Statement	Result
`select logbeta(5,3);`	`-4.6539603501575`

See Also

Functions:

- "BETA Function" on page 292

LOGCDF Function

Returns the logarithm of a left cumulative distribution function.

Categories: CAS
Probability

See: "CDF Function" on page 312

Syntax

LOGCDF(*'distribution'*, *quantile* [, *parameter-1*, ..., *parameter-k*])

Arguments

distribution

is a character constant, variable, or expression that identifies the distribution.

Note: The arguments for each of the LOGCDF distribution functions are identical to those of the corresponding CDF distribution functions.

Here are valid distributions:

Distribution	Argument
Bernoulli	'BERNOULLI'
Beta	'BETA'
Binomial	'BINOMIAL'
Cauchy	'CAUCHY'
Chi-Square	'CHISQUARE'
Conway-Maxwell-Poisson	'CONMAXPOI'
Exponential	'EXPONENTIAL'
F	'F'
Gamma	'GAMMA'
Generalized Poisson	'GENPOISSON'
Geometric	'GEOMETRIC'
Hypergeometric	'HYPERGEOMETRIC'
Laplace	'LAPLACE'
Logistic	'LOGISTIC'
Lognormal	'LOGNORMAL'
Negative binomial	'NEGBINOMIAL'
Normal	'NORMAL'|'GAUSS'

Distribution	Argument
Normal mixture	'NORMALMIX'
Pareto	'PARETO'
Poisson	'POISSON'
T	'T'
Tweedie	'TWEEDIE'
Uniform	'UNIFORM'
Wald (inverse Gaussian)	'WALD'|'IGAUSS'
Weibull	'WEIBULL'

Note: Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

quantile
is a numeric variable, constant, or expression that specifies the value of a random variable.

Data type DOUBLE

parameter-1, ..., parameter-k
are optional *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

Data type DOUBLE

Details

The LOGCDF function computes the logarithm of a left cumulative distribution function (logarithm of the left side) from various continuous and discrete distributions. For more information, see the individual distributions in the table above.

For more information about the distributions that are listed in the table, see "CDF Function" on page 312.

See Also

Functions:

- “CDF Function” on page 312
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “PDF Function” on page 693
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

LOGISTIC Function

Returns the logistic transformation of the argument.

Categories: CAS
Mathematical

Syntax

LOGISTIC(*argument*)

Arguments

argument
is a numeric variable, constant, or expression that specifies the value of a numeric random variable. When *argument* is a missing or null value, the LOGISTIC function returns a missing or null value.

Details

The LOGISTIC function returns the logistic transformation of an argument. It is typically used to convert a log odds value to a value on the probability scale. The function is mathematically expressed by the following equation:

$$logistic = \frac{e^x}{1 + e^x}$$

If the argument contains a missing value, then the LOGISTIC function returns a missing or null value.

Example

The following statement illustrates the LOGISTIC function:

Statements	Results
`select LOGISTIC(.5);`	`0.622459`
`select LOGISTIC(7.3);`	`0.999325`

LOGPDF Function

Computes the logarithm of the probability density (mass) function from various continuous and discrete distributions.

Categories: CAS
Probability

Alias: LOGPMF

See: "PDF Function" on page 693

Syntax

LOGPDF(*'distribution'*, *quantile*, *parameter-1, ..., parameter-k*)

Arguments

distribution
is a character constant, variable, or expression that identifies the distribution.

Note: The arguments for each of the LOGPDF distribution functions are identical to those of the corresponding PDF distribution functions.

Here are valid distributions:

Distribution	Argument
Bernoulli	`'BERNOULLI'`
Beta	`'BETA'`
Binomial	`'BINOMIAL'`
Cauchy	`'CAUCHY'`
Chi-Square	`'CHISQUARE'`
Conway-Maxwell-Poisson	`'CONMAXPOI'`

Distribution	Argument
Exponential	'EXPONENTIAL'
F	'F'
Gamma	'GAMMA'
Generalized Poisson	'GENPOISSON'
Geometric	'GEOMETRIC'
Hypergeometric	'HYPERGEOMETRIC'
Laplace	'LAPLACE'
Logistic	'LOGISTIC'
Lognormal	'LOGNORMAL'
Negative binomial	'NEGBINOMIAL'
Normal	'NORMAL'|'GAUSS'
Normal mixture	'NORMALMIX'
Pareto	'PARETO'
Poisson	'POISSON'
T	'T'
Tweedie	'TWEEDIE'
Uniform	'UNIFORM'
Wald (inverse Gaussian)	'WALD'|'IGAUSS'
Weibull	'WEIBULL'

Note: Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

quantile

is a numeric constant, variable, or expression that specifies the value of a random variable.

Data type DOUBLE

parameter-1, ..., parameter-k
are optional *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

Data type DOUBLE

Details

The LOGPDF function computes the logarithm of the probability density (mass) function from various continuous and discrete distributions. For more information, see the individual distributions in the table above.

For more information about the distributions that are listed in the table, see "PDF Function" on page 693.

See Also

Functions:

- "CDF Function" on page 312
- "LOGCDF Function" on page 622
- "LOGSDF Function" on page 628
- "PDF Function" on page 693
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

LOGSDF Function

Returns the logarithm of a survival function.

Categories: CAS
Probability

See: "SDF Function" on page 801

Syntax

LOGSDF(*'distribution'*, *quantile*, *parameter-1, ..., parameter-k*)

Arguments

distribution

is a character constant, variable, or expression that identifies the distribution.

Note: The arguments for each of the LOGSDF distribution functions are identical to those of the corresponding CDF distribution functions.

Here are valid distributions:

Distribution	Argument
Bernoulli	'BERNOULLI'
Beta	'BETA'
Binomial	'BINOMIAL'
Cauchy	'CAUCHY'
Chi-Square	'CHISQUARE'
Conway-Maxwell-Poisson	'CONMAXPOI'
Exponential	'EXPONENTIAL'
F	'F'
Gamma	'GAMMA'
Generalized Poisson	'GENPOISSON'
Geometric	'GEOMETRIC'
Hypergeometric	'HYPERGEOMETRIC'
Laplace	'LAPLACE'
Logistic	'LOGISTIC'
Lognormal	'LOGNORMAL'
Negative binomial	'NEGBINOMIAL'
Normal	'NORMAL'|'GAUSS'
Normal mixture	'NORMALMIX'
Pareto	'PARETO'

Distribution	Argument
Poisson	'POISSON'
T	'T'
Tweedie	'TWEEDIE'
Uniform	'UNIFORM'
Wald (inverse Gaussian)	'WALD'|'IGAUSS'
Weibull	'WEIBULL'

Note: Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

quantile
is a numeric constant, variable, or expression that specifies the value of a random variable.

Data type DOUBLE

parameter-1, ..., parameter-k
are optional *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

Data type DOUBLE

Details

The LOGSDF function computes the logarithm of the survival function from various continuous and discrete distributions. For more information, see "SDF Function" on page 801.

For more information about the distributions that are listed in the table, see "CDF Function" on page 312.

See Also

Functions:

- "CDF Function" on page 312
- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626
- "PDF Function" on page 693

- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

LOWCASE Function

Converts all letters in a character expression to lowercase.

Categories:	CAS Character
Alias:	LOWER
Returned data type:	CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

LOWCASE(*expression*)

Arguments

expression

specifies any valid expression that evaluates or can be coerced to a character string.

Requirement	Literal character expressions must be enclosed in single quotation marks.
Data type	CHAR, NCHAR, NVARCHAR, VARCHAR
See	"<sql-expression> Expression" on page 894 "FedSQL Expressions" on page 46

Details

The LOWCASE function copies a character expression, converts all uppercase letters to lowercase letters, and returns the altered value as a result.

Comparisons

The UPCASE function converts all letters in an argument to uppercase letters. The LOWCASE function converts all letters in an argument to lowercase letters.

Example

The following statement illustrates the LOWCASE function:

Statements	Results
`select lowcase('INTRODUCTION');`	`introduction`

See Also

Functions:

- "UPCASE Function" on page 847

MAD Function

Returns the median absolute deviation from the median.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

MAD(*expression-1*[, …*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value of which the median absolute deviation from the median is to be computed.

Data type DOUBLE

Details

If all arguments have missing or null values, the result is a missing or null value. Otherwise, the result is the median absolute deviation from the median of the nonmissing or non-null values. The formula for the median is the same as the one

that is used in the UNIVARIATE procedure. For more information, see Base SAS Procedures Guide: Statistical Procedures.

Example

The following statement illustrates the MAD function:

Statements	Results
`select mad(2, 4, 1, 3, 5, 999999);`	`1.5`

See Also

Functions:

- "IQR Function" on page 602
- "MEDIAN Function" on page 646
- "PCTL Function" on page 691

MAKEDATE Function

Returns the date as year, month, and day.

Categories:	Date and Time CAS
Returned data type:	DATE

Syntax

MAKEDATE(*y*, *m*, *d*)

Arguments

y
specifies the year.

m
specifies the month.

d
specifies the day.

Example

The following statements illustrates the MAKEDATE function:

Statements	Results
`select makedate(2013, 10, 30);`	30OCT2013
`select makedate(y, m, d) from adate;`	30OCT2013

See Also

Functions:

- "DATEPART Function" on page 418
- "MAKETIME Function" on page 634
- "MAKETIMESTAMP Function" on page 635
- "TIMEPART Function" on page 831

MAKETIME Function

Returns the time as hours, minutes, and seconds.

Categories:	Date and Time CAS
Returned data type:	TIME

Syntax

MAKETIME(*h*, *m*, *s*)

Arguments

h
specifies the hour.

m
specifies the minute.

s
specifies the second.

Example

The following statement illustrates the MAKETIME function:

Statements	Results
`select maketime(15,39,10);`	`15:39:10`
`select maketime(h, mn, s) from adate;`	`15:39:10`

See Also

Functions:

- “DATEPART Function” on page 418
- “MAKEDATE Function” on page 633
- “MAKETIMESTAMP Function” on page 635
- “TIMEPART Function” on page 831

MAKETIMESTAMP Function

Returns the timestamp.

Categories:	Date and Time CAS
Returned data type:	timestamp

Syntax

MAKETIMESTAMP(*d,t*)

MAKETIMESTAMP(*y,m,d,h,m,s*)

Arguments

d,t

d
specifies the date.

t
specifies the time.

y,m,d,h,m,s

y
specifies the year.

m
specifies the month.

d
specifies the day.

h
specifies the hours.

m
specifies the minutes.

s
specifies the seconds.

Example

The following statements illustrate the MAKETIMESTAMP function:

Statements	Results
`select maketimestamp(date '2013-10-30', time '10:39:10');`	`30OCT2013:10:39:10`
`select maketimestamp(2013,10,30,13,39,10);`	`30OCT2013:13:39:10`
`select maketimestamp(makedate(2013, 10, 30), maketime(13, 39, 10));`	`30OCT2013:13:39:10`

See Also

Functions:

- “DATEPART Function” on page 418
- “MAKEDATE Function” on page 633
- “MAKETIME Function” on page 634
- “TIMEPART Function” on page 831

MARGRCLPRC Function

Calculates call prices for European options on stocks, based on the Margrabe model.

Categories: CAS

Financial

Returned data type: DOUBLE

Syntax

MARGRCLPRC(*X_1*, *t*, *X_2*, *sigma1*, *sigma2*, *rho12*)

Arguments

X_1

is a nonmissing, positive value that specifies the price of the first asset.

Requirement Specify X_1 and X_2 in the same units.

Data type DOUBLE

t

is a nonmissing value that specifies the time to expiration, in years.

Data type DOUBLE

X_2

is a nonmissing, positive value that specifies the price of the second asset.

Requirement Specify X_2 and X_1 in the same units.

Data type DOUBLE

sigma1

is a nonmissing, positive fraction that specifies the volatility of the first asset.

Data type DOUBLE

sigma2

is a nonmissing, positive fraction that specifies the volatility of the second asset.

Data type DOUBLE

rho12

specifies the correlation between the first and second assets, $\rho_{x_1 x_2}$.

Range between –1 and 1

Data type DOUBLE

Details

The MARGRCLPRC function calculates the call price for European options on stocks, based on the Margrabe model. The function is based on the following relationship:

$$\text{CALL} = X_1 N(d_1) - X_2 N(d_2)$$

Arguments

X_1
specifies the price of the first asset.

X_2
specifies the price of the second asset.

N
specifies the cumulative normal density function.

$$d_1 = \frac{\left(\ln\left(\frac{N_1}{N_2}\right) + \left(\frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

$$\sigma^2 = \sigma_{x_1}^2 + \sigma_{x_2}^2 - 2\rho_{x_1,x_2}\sigma_{x_1}\sigma_{x_2}$$

The following arguments apply to the preceding equation:

t
specifies the time to expiration.

$\sigma_{x_1}^2$
specifies the variance of the first asset.

$\sigma_{x_2}^2$
specifies the variance of the second asset.

σ_{x_1}
specifies the volatility of the first asset.

σ_{x_2}
specifies the volatility of the second asset.

ρ_{x_1,x_2}
specifies the correlation between the first and second assets.

For the special case of *t*=0, the following equation is true:

$$\text{CALL} = \max((X_1 - X_2), 0)$$

Note: This function assumes that there are no dividends from the two assets.

For information about the basics of pricing, see "Using Pricing Functions" in *SAS Functions and CALL Routines: Reference*.

Comparisons

The MARGRCLPRC function calculates the call price for European options on stocks, based on the Margrabe model. The MARGRPTPRC function calculates the put price for European options on stocks, based on the Margrabe model. These functions return a scalar value.

Example

The following statements illustrate the MARGRCLPRC function:

Statements	Results
`select margrclprc(15, .5, 13, .06, .05, 1);`	2
`select margrclprc(2, .25, 1, .3, .2, 1);`	1

See Also

Functions:

- "MARGRPTPRC Function" on page 639

MARGRPTPRC Function

Calculates put prices for European options on stocks, based on the Margrabe model.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

MARGRPTPRC(X_1, *t*, X_2, *sigma1*, *sigma2*, *rho12*)

Arguments

X_1

is a nonmissing, positive value that specifies the price of the first asset.

Requirement Specify X_1 and X_2 in the same units.

Data type DOUBLE

t

is a nonmissing value that specifies the time to expiration, in years.

Data type DOUBLE

X_2

is a nonmissing, positive value that specifies the price of the second asset.

Requirement Specify X_2 and X_1 in the same units.

Data type DOUBLE

sigma1

is a nonmissing, positive fraction that specifies the volatility of the first asset.

Data type DOUBLE

sigma2

is a nonmissing, positive fraction that specifies the volatility of the second asset.

Data type DOUBLE

rho12

specifies the correlation between the first and second assets, $\rho_{x_1 y_1}$.

Range between –1 and 1

Data type DOUBLE

Details

The MARGRPTPRC function calculates the put price for European options on stocks, based on the Margrabe model. The function is based on the following relationship:

$$\text{PUT} = X_2 N(pd_1) - X_1 N(pd_2)$$

Arguments

X_1

specifies the price of the first asset.

X_2

specifies the price of the second asset.

N

specifies the cumulative normal density function.

$$pd_1 = \frac{\left(\ln\left(\frac{N_1}{N_2}\right) + \left(\frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$pd_2 = pd_1 - \sigma\sqrt{t}$$

$$\sigma^2 = \sigma_{x_1}^2 + \sigma_{x_2}^2 - 2\rho_{x_1,\,x_2}\sigma_{x_1}\sigma_{x_2}$$

The following arguments apply to the preceding equation:

`t`
is a nonmissing value that specifies the time to expiration, in years.

$\sigma_{x_1}^2$
specifies the variance of the first asset.

$\sigma_{x_2}^2$
specifies the variance of the second asset.

σ_{x_1}
specifies the volatility of the first asset.

σ_{x_2}
specifies the volatility of the second asset.

$\rho_{x_1,\,x_2}$
specifies the correlation between the first and second assets.

To view the corresponding CALL relationship, see the "MARGRCLPRC Function" on page 636.

For the special case of *t*=0, the following equation is true:

$$\text{PUT} = \max((X_2 - X_1), 0)$$

Note: This function assumes that there are no dividends from the two assets.

For information about the basics of pricing, see "Using Pricing Functions" in *SAS Functions and CALL Routines: Reference*.

Comparisons

The MARGRPTPRC function calculates the put price for European options on stocks, based on the Margrabe model. The MARGRCLPRC function calculates the call price for European options on stocks, based on the Margrabe model. These functions return a scalar value.

Example

The following statements illustrate the MARGRPTPRC function:

Statements	Results
`select margrptprc(2, .25, 3, .06, .2, 1);`	`1.0000000000973`
`select margrptprc(3, .25, 4, .05, .3, 1);`	`1.00157624907712`

See Also

Functions:

- "MARGRCLPRC Function" on page 636

MAX Function

Returns the maximum value in a column.

Categories: Aggregate
Descriptive Statistics
CAS

Returned data type: The same data type as the expression

Syntax

MAX([DISTINCT] *expression*)

Arguments

expression
specifies any valid SQL expression.

Data type All data types are supported.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

DISTINCT
(optional) eliminates duplicate rows from the aggregate calculation.

Details

The MAX function ignores null values and SAS missing values.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Comparisons

The MIN function returns the minimum value in a column. The MAX function returns the maximum value in a column.

Example

Table: DENSITIES on page 1173

The following statement illustrates the MAX function.

Statements	Results
`select max(population) from densities;`	`34248705`

See Also

Functions:

- “MIN Function” on page 648

SELECT Statement Clauses:

- “SELECT Clause” on page 956
- “GROUP BY Clause” on page 968
- “HAVING Clause” on page 970

MDY Function

Returns a SAS date value from month, day, and year values.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

MDY(*month*, *day*, *year*)

Arguments

month
specifies a numeric expression that represents a whole number from 1 through 12.

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

day
specifies a numeric expression that represents a whole number from 1 through 31.

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

year
specifies a numeric expression that represents a two-digit or four-digit year. The YEARCUTOFF= system option defines the year value for two-digit dates.

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

Example

The following statements illustrate the MDY function:

Statements	Results
`select put(mdy(8,27,12),date7.);`	27AUG12
`select put(mdy(8,27,90),date7.);`	27AUG90

See Also

Concepts:

- "Dates and Times in FedSQL" on page 58

Functions:

- "DAY Function" on page 420
- "MONTH Function" on page 655
- "YEAR Function" on page 864

MEAN Function

Returns the arithmetic mean (average) of the non-null or nonmissing arguments.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

MEAN(*expression-1*[, …*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement At least one non-null or nonmissing argument is required. Otherwise, the function returns a null or missing value.

Data type	DOUBLE
See	“<sql-expression> Expression” on page 894
	“FedSQL Expressions” on page 46

Comparisons

The GEOMEAN function returns the geometric mean, the HARMEAN function returns the harmonic mean, whereas the MEAN function returns the arithmetic mean (average).

Example

The following statements illustrate the MEAN function:

Statements	Results
`select mean(2,.,.,6);`	`4`
`select mean(1,2,3,2);`	`2`

See Also

Functions:

- “GEOMEAN Function” on page 537
- “GEOMEANZ Function” on page 539
- “HARMEAN Function” on page 540
- “HARMEANZ Function” on page 542
- “MEDIAN Function” on page 646

MEDIAN Function

Returns the median value.

Categories:	CAS Descriptive Statistics
Returned data type:	DOUBLE

Syntax

MEDIAN(*expression-1*[, …*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The MEDIAN function returns the median of the nonmissing or nonnull values. If all arguments have missing or null values, the result is a missing or null value.

Note: The formula that is used in the MEDIAN function is the same as the formula that is used in PROC UNIVARIATE in Base SAS Procedures Guide: Statistical Procedures. For more information, see SAS Elementary Statistics Procedures.

Comparisons

The MEDIAN function returns the median of nonmissing or nonnull values, whereas the MEAN function returns the arithmetic mean (average).

Example

The following statements illustrate the MEDIAN function:

Statements	Results
`select median(2, 4, 1, 3);`	2.5
`select median(5, 8, 0, 3, 4);`	4

See Also

Functions:

- “MEAN Function” on page 645

MIN Function

Returns the minimum value in an expression.

Categories:	Aggregate Descriptive Statistics CAS
Returned data type:	The same data type as the expression

Syntax

MIN([DISTINCT] *expression*)

Arguments

expression
specifies any valid SQL expression.

Data type All data types are supported.

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

DISTINCT
(optional) eliminates duplicate rows from the aggregate calculation.

Details

The MIN function ignores null values and SAS missing values.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Comparisons

The MAX function returns the maximum value in a column. The MIN function returns the minimum value in a column.

Example

Table: DENSITIES on page 1173

The following statement illustrates the MIN function.

Statements	Results
`select min(density) from densities;`	`6.154657`

See Also

Functions:

- "MAX Function" on page 642

SELECT Statement Clauses:

- "SELECT Clause" on page 956
- "GROUP BY Clause" on page 968
- "HAVING Clause" on page 970

MINUTE Function

Returns the minute from a time or datetime value.

Categories:	Date and Time CAS
Returned data type:	TINYINT

Syntax

MINUTE(*time* | *datetime*)

Arguments

time

specifies any valid expression that represents a time value.

Data type TIME

See "Overview of Expressions and Predicates" on page 873

datetime

specifies any valid expression that represents a datetime value.

Data type TIMESTAMP

See "Overview of Expressions and Predicates" on page 873

Example

Table: CUSTONLINE on page 1172

The following statement illustrates the MINUTE function:

Statements	Results
`select minute(endtime) from custonline;`	5 21 4 25 47 6 5 15 35 6
`select minute(current_time);`	16
`select minute(localtimestamp);`	16

See Also

- "Dates and Times in FedSQL" on page 58

Functions:

- "DAY Function" on page 420
- "HOUR Function" on page 549
- "MONTH Function" on page 655
- "SECOND Function" on page 806

- “YEAR Function” on page 864

MOD Function

Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid most unexpected floating-point results.

Categories:	CAS Mathematical
Returned data type:	DOUBLE

Syntax

MOD(*dividend-expression*, *divisor-expression*)

Arguments

dividend-expression

specifies a dividend that is any valid expression that evaluates to a numeric value.

Data type	DOUBLE
See	“<sql-expression> Expression” on page 894 “FedSQL Expressions” on page 46

divisor-expression

specifies a divisor that is any valid expression that evaluates to a numeric value.

Restriction	*divisor-expression* cannot be 0
Data type	DOUBLE
See	“<sql-expression> Expression” on page 894 “FedSQL Expressions” on page 46

Details

The MOD function returns the remainder from the division of *dividend-expression* by *divisor-expression*. When the result is nonzero, the result has the same sign as the first argument. The sign of the second argument is ignored.

The computation that is performed by the MOD function is exact if both of the following conditions are true:

- Both arguments are exact integers.
- All integers that are less than either argument have exact 8-byte floating-point representations.

If either of the above conditions is not true, a small amount of numerical error can occur in the floating-point computation. In this case, the following occurs.

- MOD returns zero if the remainder is very close to zero or very close to the value of the second argument.
- MOD returns a null or missing value if the remainder cannot be computed to a precision of approximately three digits or more. In this case, SAS also writes an error message to the log.

Comparisons

Here are some comparisons between the MOD and MODZ functions:

- The MOD function performs extra computations, called fuzzing, to return an exact zero when the result would otherwise differ from zero because of numerical error.
- The MODZ function performs no fuzzing.
- Both the MOD and MODZ functions return a null or missing value if the remainder cannot be computed to a precision of approximately three digits or more.

Example

The following statements illustrate the MOD function:

Statements	Results
`select mod(10,3);`	1
`select mod(.35,-.1);`	0.05
`select mod(17,3);`	2
`select mod(.3,-.9);`	0.3

See Also

Functions:

- “MODZ Function” on page 653
- “INTZ Function” on page 598

MODZ Function

Returns the remainder from the division of the first argument by the second argument, using zero fuzzing.

Categories:	CAS Mathematical
Returned data type:	DOUBLE

Syntax

MODZ(*dividend-expression*, *divisor-expression*)

Arguments

dividend-expression
specifies a dividend that is any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

divisor-expression
specifies a divisor that is any valid expression that evaluates to a numeric value.

Restriction *divisor-expression* cannot be 0

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The MODZ function returns the remainder from the division of *dividend-expression* by *divisor-expression*. When the result is nonzero, the result has the same sign as the first argument. The sign of the second argument is ignored.

The computation that is performed by the MODZ function is exact if both of the following conditions are true:

- Both arguments are exact integers.

- All integers that are less than either argument have exact 8-byte floating-point representation.

If either of the above conditions is not true, a small amount of numerical error can occur in the floating-point computation. For example, when you use exact arithmetic and the result is zero, MODZ might return a very small positive value or a value slightly less than the second argument.

Comparisons

Here are some comparisons between the MODZ and MOD functions:

- The MODZ function performs no fuzzing.
- The MOD function performs fuzzing, to return an exact zero when the result would otherwise differ from zero because of numerical error.
- Both the MODZ and MOD functions return a null or missing value if the remainder cannot be computed to a precision of approximately three digits or more.

Examples

Example 1

The following statements illustrate the MOD function:

Statements	Results
`select mod(10,3);`	1
`select mod(.35,-.1);`	0.05
`select mod(17,3);`	2
`select mod(.3,-.9);`	0.3

Example 2

The following statements illustrate the MODZ function:

Statements	Results
`select modz(10,3);`	1

Statements	Results
`select modz(.35,-.1);`	`0.05`
`select modz(17,3);`	`2`
`select modz(.3,-.9);`	`0.3`

See Also

Functions:

- "INTZ Function" on page 598
- "MOD Function" on page 651

MONTH Function

Returns the numeric month from a date or datetime value.

Categories:	Date and Time CAS
Returned data type:	TINYINT

Syntax

MONTH(*date* | *datetime*)

Arguments

date

specifies any valid expression that represents a date value.

Data type DATE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

datetime

specifies any valid expression that represents a datetime value.

Data type TIMESTAMP

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Example

Table: CUSTONLINE on page 1172

The following statement illustrates the MONTH function:

Statements	Results
`select month(endtime) from custonline;`	9 10 10 11 12 1 1 2 3 3
`select month(current_time);`	10

See Also

- "Dates and Times in FedSQL" on page 58

Functions:

- "DAY Function" on page 420
- "HOUR Function" on page 549
- "MINUTE Function" on page 649
- "SECOND Function" on page 806
- "YEAR Function" on page 864

MORT Function

Returns amortization parameters.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

MORT(*a*, *p*, *r*, *n*)

Arguments

a

specifies any valid expression that evaluates to the initial amount.

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

p

specifies any valid expression that evaluates to the periodic payment.

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

r

specifies any valid expression that evaluates to the periodic interest rate that is expressed as a fraction.

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

n

specifies any valid expression that evaluates to the number of compounding periods.

Range $n \geq 0$

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

Details

Calculating Results

The MORT function returns the missing argument in the list of four arguments from an amortization calculation with a fixed interest rate that is compounded each period. The arguments are related by the following equation:

$$p = \frac{ar(1+r)^n}{(1+r)^n - 1}$$

One missing argument must be provided. The value is then calculated from the remaining three. No adjustment is made to convert the results to round numbers.

Restrictions in Calculating Results

The MORT function returns an invalid argument note to the SAS log and sets _ERROR_ to 1 if one of the following argument combinations is true:

- rate < –1 or n < 0
- principal <= 0 or payment <= 0 or n <= 0
- principal <= 0 or payment <= 0 or rate <= –1
- principal * rate > payment
- principal > payment * n

Example

In the following program, an amount of $50,000 is borrowed for 30 years at an annual interest rate of 10% compounded monthly.

```
select mort(50000, . , .10/12, 30*12);
```

The value that is returned is 438.79 (rounded). The second argument is set to missing, which indicates that the periodic payment is to be calculated. The 10% nominal annual rate has been converted to a monthly rate of 0.10/12. The rate is the fractional (not the percentage) interest rate per compounding period. The 30 years are converted to 360 months.

N Function

Returns the number of non-null or nonmissing numeric values.

Categories:	CAS Special
Returned data type:	INTEGER

Syntax

N(*expression* [, …*expression*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement At least one argument is required.

Data type DECIMAL, DOUBLE, NUMERIC

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

Null values are converted to missing values and are counted as missing values.

Comparisons

The N function counts non-null and nonmissing values, whereas the NMISS function counts missing values. The N function requires numeric arguments.

Example

The following statements illustrate the N function:

Statements	Results
`select n(1,0,.,2,5,.);`	4
`select n(1,2);`	2

See Also

Functions:

- "NMISS Function" on page 661.

NETPV Function

Returns the net present value as a percent.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

NETPV(*r*, *freq*, *c0, c1, ..., cn*)

Arguments

r

is numeric, the interest rate over a specified base period of time expressed as a fraction.

Range *r* >= 0

Data type DOUBLE

freq

is numeric, the number of payments during the base period of time that is specified with the rate *r*.

Range *freq* > 0

Data type DOUBLE

Note The case *freq* = 0 is a flag to allow continuous discounting.

c0, c1 ,..., cn

are numeric cash flows that represent cash outlays (payments) or cash inflows (income) occurring at times 0, 1, ...n. These cash flows are assumed to be equally spaced, beginning-of-period values. Negative values represent payments, positive values represent income, and values of 0 represent no cash flow at a given time. The *c0* argument and the *c1* argument are required.

Data type DOUBLE

Details

The NETPV function returns the net present value at time 0 for the set of cash payments *c0*,*c1*, ...,*cn*, with a rate *r* over a specified base period of time. The

argument *freq*>0 describes the number of payments that occur over the specified base period of time.

The net present value is given by the equation:

$$\mathrm{NETPV}(r, freq, c_0, c_1, ..., c_n) = \sum_{i=0}^{n} c_i x^i$$

The following relationship applies to the preceding equation:

$$x = \begin{cases} \frac{1}{(1+r)^{(1/freq)}} & freq > 0 \\ \varepsilon^{-r} & freq = 0 \end{cases}$$

Missing values in the payments are treated as 0 values. When *freq*>0, the rate *r* is the effective rate over the specified base period. To compute with a quarterly rate (the base period is three months) of 4% with monthly cash payments, set *freq* to 3 and set *r* to .04.

If *freq* is 0, continuous discounting is assumed. The base period is the time interval between two consecutive payments, and the rate *r* is a nominal rate.

To compute with a nominal annual interest rate of 11% discounted continuously with monthly payments, set *freq* to 0 and set *r* to .11/12.

Example

For an initial investment of $500 that returns biannual payments of $200, $300, and $400 over the succeeding 6 years and an annual discount rate of 10%, the net present value of the investment can be expressed as follows:

```
select netpv(.10,.5,-500,200,300,400);
```

The value that is returned is 95.982864829.

See Also

Functions:

- "NPV Function" on page 685

NMISS Function

Returns the number of null values or SAS missing values in an expression.

Categories: Aggregate
Scalar
CAS

Syntax

Aggregate:

NMISS([DISTINCT] *expression*)

Scalar:

NMISS(*expression*[, ...*expression*])

Arguments

expression (aggregate)

specifies a SQL expression that evaluates to a character column.

Data type	CHAR
Returned data type	BIGINT

expression (scalar)

specifies a SQL expression that evaluates to a numeric value. Multiple expressions must be separated by commas.

Data type	DECIMAL, DOUBLE, NUMERIC
Returned data type	INTEGER
See	“<sql-expression> Expression” on page 894
	“FedSQL Expressions” on page 46

DISTINCT

(optional) eliminates duplicate rows from the aggregate calculation.

Details

NMISS returns the number of null or SAS missing values. When FedSQL is run in ANSI mode, ANSI null values are converted to SAS missing values.

When used as an aggregate function, NMISS returns the number of null or missing values found in a single character column. You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

When used as a scalar function, NMISS returns the number of null or missing values among the specified expression(s).

Comparisons

The NMISS function returns the number of null or SAS missing values. The N function returns the number of non-null and non-missing values.

Examples

Example 1: Aggregate Operation

Table: WORLDTEMPS on page 1184

The following statement illustrates the use of NMISS as an aggregate function.

Statements	Results
`select nmiss(AvgHigh) from worldtemps;`	1

Example 2: Scalar Operation

The following statement illustrates the use of NMISS as a scalar function:

Statements	Results
`select nmiss(1,2,.,3,.);`	2

See Also

- "How FedSQL Processes Nulls and SAS Missing Values" on page 20

Functions:

- "N Function" on page 658

SELECT Statement Clauses:

- "SELECT Clause" on page 956
- "GROUP BY Clause" on page 968
- "HAVING Clause" on page 970

NOMRATE Function

Returns the nominal annual interest rate.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

NOMRATE(*compounding-interval*, *rate*)

Arguments

compounding-interval
is a SAS interval. This value represents how often the returned value is compounded.

Data type CHAR

rate
is numeric. *Rate* is the effective annual interest rate (expressed as a percentage) that is compounded at each interval.

Data type DOUBLE

Details

The NOMRATE function returns the nominal annual interest rate. NOMRATE computes the nominal annual interest rate that corresponds to an effective annual interest rate.

The following details apply to the NOMRATE function:

- The values for rates must be at least –99.
- In considering an effective interest rate and a compounding interval, if *compounding-interval* is 'CONTINUOUS', then the value that is returned by NOMRATE equals $\log_e(1+[rate/100])$.

  If *compounding-interval* is not 'CONTINUOUS', and *m* intervals occur in a year, the value that is returned by NOMRATE equals the following:

  $$m\left((1+\frac{rate}{100})^{\frac{1}{m}}-1\right)$$

- The following values are valid for *compounding-interval*:

- 'CONTINUOUS'
- 'DAY'
- 'SEMIMONTH'
- 'MONTH'
- 'QUARTER'
- 'SEMIYEAR'
- 'YEAR'

- If the interval is 'DAY', then *m*=365.

Example

- If an effective rate is 10% when compounded monthly, the corresponding nominal rate can be expressed as follows:

```
select NOMRATE('MONTH', 10);
```

```
effective_rate1=9.56896851468451
```

- If an effective rate is 10% when compounded quarterly, the corresponding nominal rate can be expressed as follows:

```
select NOMRATE('QUARTER', 10);
```

```
effective_rate2=9.64547563377804
```

NOTALNUM Function

Searches a character string for a non-alphanumeric character, and returns the first character position at which the character is found.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

NOTALNUM('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

start

specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the NOTALNUM function depend directly on the translation table that is in effect (see “TRANTAB= System Option” in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTALNUM function searches a string for the first occurrence of any character that is not a digit or an uppercase or lowercase letter. If such a character is found, NOTALNUM returns the position in the string of that character. If no such character is found, NOTALNUM returns a value of 0.

If you use only one argument, NOTALNUM begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTALNUM returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTALNUM function searches a character string for a non-alphanumeric character. The ANYALNUM function searches a character string for an alphanumeric character.

Example

The following statements illustrate the NOTALNUM function:

Statements	Results
`select notalnum('abc123, x');`	`7 (position of comma)`
`select notalnum('+abcd');`	`1 (position of plus sign)`
`select notalnum('789ab');`	`0`
`select notalnum('myname@gmail.com');`	`7 (position of @ )`
`select notalnum('531-66-1234');`	`4 (position of first dash)`
`select notalnum('531-66-1234', -99);`	`7 (position of last dash in string ) */`

See Also

Functions:

- "ANYALNUM Function" on page 261

NOTALPHA Function

Searches a character string for a nonalphabetic character, and returns the first character position at which the character is found.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

NOTALPHA('*expression*'[, *start*])

Arguments

expression

specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

start

specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the NOTALPHA function depend directly on the translation table that is in effect (see “TRANTAB= System Option” in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTALPHA function searches a string for the first occurrence of any character that is not an uppercase or lowercase letter. If such a character is found, NOTALPHA returns the position in the string of that character. If no such character is found, NOTALPHA returns a value of 0.

If you use only one argument, NOTALPHA begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTALPHA returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTALPHA function searches a character string for a nonalphabetic character. The ANYALPHA function searches a character string for an alphabetic character.

Example

The following statements illustrate the NOTALPHA function:

Statements	Results
`select notalpha('123abc456');`	`1 (position of 1)`
`select notalpha('abcd1234');`	`5 (position of 1)`
`select notalpha('abcd');`	`0`
`select notalpha('789abc456',-99);`	`9 (position of 6)`
`select notalpha('abc$123');`	`4 (position of dollar sign)`

See Also

Functions:

- "ANYALPHA Function" on page 263

NOTDIGIT Function

Searches a character string for any character that is not a digit, and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

NOTDIGIT('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the NOTDIGIT function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTDIGIT function searches a string for the first occurrence of any character that is not a digit. If such a character is found, NOTDIGIT returns the position in the string of that character. If no such character is found, NOTDIGIT returns a value of 0.

If you use only one argument, NOTDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTDIGIT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTDIGIT function searches a character string for any character that is not a digit. The ANYDIGIT function searches a character string for a digit.

Example

The following statements illustrate the NOTDIGIT function:

Statements	Results
`select notdigit('123abc456');`	`4 (position of a)`
`select notdigit('1234');`	`0`
`select notdigit('123abc456',-99);`	`6 (position of last non-digit in string)`
`select notdigit('$12,300');`	`1 (position of dollar sign)`
`select notdigit('$12,300', -99);`	`4 (position of comma)`

See Also

Functions:

- "ANYDIGIT Function" on page 265

NOTFIRST Function

Searches a character string for an invalid first character in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

NOTFIRST('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

start

specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The NOTFIRST function does not depend on the TRANTAB, ENCODING, or LOCALE system options.

The NOTFIRST function searches a string for the first occurrence of any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7. These characters are any except the underscore (_) and uppercase or lowercase English letters. If such a character is found, NOTFIRST returns the position in the string of that character. If no such character is found, NOTFIRST returns a value of 0.

If you use only one argument, NOTFIRST begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTFIRST returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTFIRST function searches a string for the first occurrence of any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7. The ANYFIRST function searches a string for the first occurrence of any character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7.

Example

The following statements illustrate the NOTFIRST function:

Statements	Results
`select notfirst('123aBc4D6');`	`1 (position of 1)`
`select notfirst('123aBc4D6',-99);`	`9 (position of 6)`
`select notfirst('Bac123, x');`	`4 (position of 1)`
`select notfirst('x,&,y,z,20,!');`	`2 (position of ampersand)`
`select notfirst('x,&,y,z,20,!', -99);`	`12 (position of exclamation mark)`

See Also

Functions:

- "ANYFIRST Function" on page 267

NOTLOWER Function

Searches a character string for a character that is not a lowercase letter, and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

NOTLOWER('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

start

specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the NOTLOWER function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTLOWER function searches a string for the first occurrence of any character that is not a lowercase letter. If such a character is found, NOTLOWER returns the position in the string of that character. If no such character is found, NOTLOWER returns a value of 0.

If you use only one argument, NOTLOWER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTLOWER returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTLOWER function searches a character string for a character that is not a lowercase letter. The ANYLOWER function searches a character string for a lowercase letter.

Example

The following statements illustrate the NOTLOWER function:

Statements	Results
`select notlower('123aBc4D6');`	`1 (position of 1)`
`select notlower('123aBc4D6',-99);`	`9 (position of 6)`
`select notlower('DAN BROWN');`	`1 (position of D)`
`select notlower('dan brown');`	`4 (position of space)`
`select notlower('mr. dan brown');`	`3 (position of period)`

See Also

Functions:

- "ANYLOWER Function" on page 269

NOTNAME Function

Searches a character string for an invalid character in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

NOTNAME('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

start

specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The NOTNAME function does not depend on the TRANTAB, ENCODING, or LOCALE system options.

The NOTNAME function searches a string for the first occurrence of any character that is not valid in a SAS variable name under VALIDVARNAME=V7. These characters are any except underscore (_), digits, and uppercase or lowercase English letters. If such a character is found, NOTNAME returns the position in the string of that character. If no such character is found, NOTNAME returns a value of 0.

If you use only one argument, NOTNAME begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTNAME returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTNAME function searches a string for the first occurrence of any character that is not valid in a SAS variable name under VALIDVARNAME=V7. The ANYNAME function searches a string for the first occurrence of any character that is valid in a SAS variable name under VALIDVARNAME=V7.

Example

The following statements illustrate the NOTNAME function:

Statements	Results
`select notname('<&abc!');`	`1 (position of angle bracket)`
`select notname('&abc!');`	`1 (position of ampersand)`
`select notname('abc!>');`	`4 (position of exclamation mark)`
`select notname('789!>');`	`4 (position of exclamation mark)`
`select notname('abc!>', -99);`	`5 (position of angle bracket)`

See Also

Functions:

- "ANYNAME Function" on page 271

NOTPUNCT Function

Searches a character string for a character that is not a punctuation character, and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

NOTPUNCT('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

start

specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the NOTPUNCT function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTPUNCT function searches a string for the first occurrence of a character that is not a punctuation character. If such a character is found, NOTPUNCT returns the position in the string of that character. If no such character is found, NOTPUNCT returns a value of 0.

If you use only one argument, NOTPUNCT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTPUNCT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTPUNCT function searches a character string for a character that is not a punctuation character. The ANYPUNCT function searches a character string for a punctuation character.

Example

The following statements illustrate the NOTPUNCT function:

Statements	Results
`select notpunct('abc123, x');`	`1 (position of a)`
`select notpunct('!?:;,');`	`0`
`select notpunct('987&efg');`	`1 (position of 9)`

See Also

Functions:

- "ANYPUNCT Function" on page 273

NOTSPACE Function

Searches a character string for a character that is not a whitespace character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first character position at which that character is found.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

NOTSPACE('*expression*'[, *start*])

Arguments

expression

specifies any valid expression that evaluates or can be coerced to a character string.

Data type	CHAR, NCHAR, VARCHAR, NVARCHAR
See	"<sql-expression> Expression" on page 894 "FedSQL Expressions" on page 46

start

specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the NOTSPACE function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTSPACE function searches a string for the first occurrence of a character that is not a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. If such a character is found, NOTSPACE returns the position in the string of that character. If no such character is found, NOTSPACE returns a value of 0.

If you use only one argument, NOTSPACE begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTSPACE returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTSPACE function searches a character string for the first occurrence of a character that is not a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. The ANYSPACE function searches a character string for the first occurrence of a character that is a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed.

Example

The following statements illustrate the NOTSPACE function:

Statements	Results
`select notspace('abc123, x');`	`1  (position of a)`
`select notspace('Ma ry Smith');`	`1 (position of M)`

Statements	Results
`select notspace('Ma ry Smith',-99);`	`11 (position of h)`

See Also

Functions:

- "ANYSPACE Function" on page 275

NOTUPPER Function

Searches a character string for a character that is not an uppercase letter, and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

NOTUPPER('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the NOTUPPER function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTUPPER function searches a string for the first occurrence of a character that is not an uppercase letter. If such a character is found, NOTUPPER returns the position in the string of that character. If no such character is found, NOTUPPER returns a value of 0.

If you use only one argument, NOTUPPER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTUPPER returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTUPPER function searches a character string for a character that is not an uppercase letter. The ANYUPPER function searches a character string for an uppercase letter.

Example

The following statements illustrate the NOTUPPER function:

Statements	Results
`select notupper('aBc4D');`	`1 (position of a)`
`select notupper('aBc4D', -99);`	`4 (position of 4)`
`select notupper('123aBc4D6');`	`1 (position of 1)`
`select notupper('123aBc4D6', 5);`	`6 (position of c)`
`select notupper('123aBc4D6', -5);`	`4 (position of a)`

Statements	Results
`select notupper('ABCDE');`	`0`

See Also

Functions:

- "ANYUPPER Function" on page 276

NOTXDIGIT Function

Searches a character string for a character that is not a hexadecimal character, and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

NOTXDIGIT('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The NOTXDIGIT function searches a string for the first occurrence of any character that is not a digit or an uppercase or lowercase A, B, C, D, E, or F. If such a character is found, NOTXDIGIT returns the position in the string of that character. If no such character is found, NOTXDIGIT returns a value of 0.

If you use only one argument, NOTXDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTXDIGIT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTXDIGIT function searches a character string for a character that is not a hexadecimal character. The ANYXDIGIT function searches a character string for a character that is a hexadecimal character.

Example

The following statements illustrate the NOTXDIGIT function:

Statements	Results
`select notxdigit('123aBc4D6');`	`0 (no special characters)`
`select notxdigit('abcd456');`	`0 (no special characters)`
`select notxdigit('789ab!',-99);`	`6 (position of exclamation mark)`
`select notxdigit('&789ab!');`	`1 (position of ampersand )`

See Also

Functions:

- "ANYXDIGIT Function" on page 278

NPV Function

Returns the net present value with the rate expressed as a percentage.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

NPV(*r*, *freq*, *c0*, *c1*, …, *cn*)

Arguments

r

is numeric, the interest rate over a specified base period of time expressed as a percentage.

Data type DOUBLE

freq

is numeric, the number of payments during the base period of time specified with the rate *r*.

Range *freq* > 0

Data type DOUBLE

Note The case *freq* = 0 is a flag to allow continuous discounting.

c0, c1, …, cn

are numeric cash flows that represent cash outlays (payments) or cash inflows (income) occurring at times 0, 1, ...n. These cash flows are assumed to be equally spaced, beginning-of-period values. Negative values represent payments, positive values represent income, and values of 0 represent no cash flow at a given time. The *c0* argument and the *c1* argument are required.

Data type DOUBLE

Comparisons

The NPV function is identical to NETPV, except that the *r* argument is provided as a percentage.

Example

The following statement illustrates the NPV function:

Statements	Results
`select npv(1,2,3,4,5,6,7,8);`	`32.50727639`

See Also

Functions:

- "NETPV Function" on page 660

NWKDOM Function

Returns the date for the *n*th occurrence of a weekday for the specified month and year.

Categories:	CAS Date and Time
Returned data type:	DOUBLE

Syntax

NWKDOM(*n*, *weekday*, *month*, *year*)

Arguments

n

specifies the numeric week of the month that contains the specified day.

Range 1–5

Data type DOUBLE

Tip *N*=5 indicates that the specified day occurs in the last week of that month. Sometimes *n*=4 and *n*=5 produce the same results.

weekday
specifies the number that corresponds to the day of the week.

Range 1–7

Data type DOUBLE

Tip Sunday is considered the first day of the week and has a *weekday* value of 1.

month
specifies the number that corresponds to the month of the year.

Range 1–12

Data type DOUBLE

year
specifies a four-digit calendar year.

Data type DOUBLE

Details

The NWKDOM function returns a SAS date value for the *n*th weekday of the month and year that you specify. Use any valid SAS date format, such as the DATE9. format, to display a calendar date. You can specify *n*=5 for the last occurrence of a particular weekday in the month.

Sometimes *n*=5 and *n*=4 produce the same result. These results occur when there are only four occurrences of the requested weekday in the month. For example, if the month of January begins on a Sunday, there are five occurrences of Sunday, Monday, and Tuesday, but only four occurrences of Wednesday, Thursday, Friday, and Saturday. In this case, specifying *n*=5 or *n*=4 for Wednesday, Thursday, Friday, or Saturday produces the same result.

If the year is not a leap year, February has 28 days and there are four occurrences of each day of the week. In this case, *n*=5 and *n*=4 produce the same results for every day.

Comparisons

In the NWKDOM function, the value for *weekday* corresponds to the numeric day of the week beginning on Sunday. This value is the same value that is used in the WEEKDAY function, where Sunday =1, and so on. The value for *month* corresponds to the numeric month of the year beginning in January. This value is the same value that is used in the MONTH function, where January =1, and so on.

You can use the NWKDOM function to calculate events that are not defined by the HOLIDAY function. For example, if a university always schedules graduation on the

first Saturday in June, then you can use the following statement to calculate the date: `UnivGrad = nwkdom(1, 7, 6, year);`

Examples

Example 1: Returning Date Values

The following statements illustrate the use of the NWKDOM function to return the date for specific occurrences of a weekday for a specified month and year. The examples apply the WEEKDATX. format to display the output SAS date values in a readable format.

Statements	Results
`select put(nwkdom(3, 2, 5, 2012), weekdatx.);`	`Monday, 21 May 2012`
`select put(nwkdom(4, 4, 11, 2012), weekdatx.);`	`Wednesday, 28 November 2012`
`select put(nwkdom(4, 7, 11, 2012), weekdatx.);`	`Saturday, 24 November 2012`
`select put(nwkdom(1, 1, 1, 2013), weekdatx.);`	`Sunday, 6 January 2013`
`select put(nwkdom(2, 3, 9, 2012), weekdatx.);`	`Tuesday, 11 September 2012`
`select put(nwkdom(5, 5, 12, 2012), weekdatx.);`	`Thursday, 27 December 2012`

Example 2: Returning a Date

The following statement illustrates the use of the NWKDOM function to return the date of the last Monday in May. The DATE9. format is applied to display the date value.

Statements	Results
`select put(nwkdom(5,2,5,2012), date9.);`	`28MAY2012`

See Also

Functions:

- “HOLIDAY Function” on page 545
- “INTNX Function” on page 582
- “WEEKDAY Function” on page 860

OCTET_LENGTH Function

Returns the number of bytes in a string of any data type.

Categories:	Character CAS
Returned data type:	BIGINT

Syntax

OCTET_LENGTH(*expression*)

Arguments

expression
specifies any valid expression.

Data type	All data types are valid.
See	"<sql-expression> Expression" on page 894 "FedSQL Expressions" on page 46

Details

The OCTET_LENGTH function counts a multibyte character as multiple characters. For example, if the specified string contains 6 two-byte characters, the OCTET_LENGTH function returns a value of 12.

If the character string is a bit string, the OCTET_LENGTH function returns the number of bytes it takes to hold the number of bits.

Comparisons

The CHARACTER_LENGTH function also returns the number of characters in a character string, but counts a multibyte character as a single character. The OCTET_LENGTH function counts a multibyte character as multiple characters.

Example

The following statements illustrate the OCTECT_LENGTH function:

Statements	Results
`select octect_length('December');`	8
`select octet_length(B'0100011100');`	2
`select octet_length(x'FEA364BEA234');`	6
`select octet_length(' FedSQL ');`	18

See Also

Functions:

- "CHARACTER_LENGTH Function" on page 361

ORDINAL Function

Orders a list of values, and returns a value that is based on a position in the list.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

ORDINAL(*position*, *expression-1, expression-2* [, …*expression-n*])

Arguments

position
specifies a whole number that is less than or equal to the number of elements in the list of arguments.

Requirement *position* must be a positive number.

Data type DOUBLE

expression
specifies any valid expression that evaluates to a numeric value.

Requirement At least two arguments are required.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The ORDINAL function sorts the list and returns the argument in the list that is specified by *position*. Missing values are sorted low and are placed before any numeric values.

Comparisons

The ORDINAL function counts both null, missing, non-null, and nonmissing values, whereas the SMALLEST function counts only non-null and nonmissing values.

Example

The following statement illustrates the ORDINAL function:

Statement	Result
`select ordinal(4,1,2,3,-4,5,6,7);`	3

PCTL Function

Returns the percentile that corresponds to the percentage.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

PCTL[*n*](*percentage*, *expression*[, …*expression*])

Arguments

n

is a digit from 1 to 5 that specifies the definition of the percentile to be computed.

Default definition 5

Data type DOUBLE

See QNTLDEF= (alias PCTLDEF=) descriptions in the table "Methods for Computing Quantile Statistics" in the *Base SAS Procedures Guide*.

percentage

specifies the percentile to be computed.

Data type DOUBLE

Tip *percentage* is numeric where, 0≤*percentage*≤100.

expression

specifies any valid expression that evaluates to a numeric value, whose value is computed in the percentile calculation.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The PCTL function returns the percentile of the non-null or nonmissing values corresponding to the percentage. If *percentage* is null or missing, less than zero, or greater than 100, the PCTL function generates an error message.

Example

The following statements illustrate the PCTL function:

Description	Statement	Result
Lower quartile	`select pctl(25,2,4,1,3);`	1.5
Percentile definition two	`select pctl2(25,2,4,1,3);`	1

Description	Statement	Result
Lower tertile	`select pctl(100/3,2,4,1,3);`	2
Percentile definition three	`select pctl3(100/3,2,4,1,3);`	2
Median	`select pctl(50,2,4,1,3);`	2.5
Upper tertile	`select pctl(200/3,2,4,1,3);`	3
Upper quartile	`select pctl(75,2,4,1,3);`	3.5

PDF Function

Returns a value from a probability density (mass) distribution.

Categories: CAS
Probability

Alias: PMF

Data type: DOUBLE

Syntax

PDF(*'distribution'*, *quantile* [, *parameter-1, ..., parameter-k*])

Arguments

distribution
is a character constant, variable, or expression that identifies the distribution. Here are valid distributions:

Distribution	Argument
Bernoulli	`'BERNOULLI'`
Beta	`'BETA'`
Binomial	`'BINOMIAL'`
Cauchy	`'CAUCHY'`
Chi-Square	`'CHISQUARE'`

Distribution	Argument
Conway-Maxwell-Poisson	'CONMAXPOI'
Exponential	'EXPONENTIAL'
F	'F'
Gamma	'GAMMA'
Generalized Poisson	'GENPOISSON'
Geometric	'GEOMETRIC'
Hypergeometric	'HYPERGEOMETRIC'
Laplace	'LAPLACE'
Logistic	'LOGISTIC'
Lognormal	'LOGNORMAL'
Negative binomial	'NEGBINOMIAL'
Normal	'NORMAL'|'GAUSS'
Normal mixture	'NORMALMIX'
Pareto	'PARETO'
Poisson	'POISSON'
T	'T'
Tweedie	'TWEEDIE'
Uniform	'UNIFORM'
Wald (inverse Gaussian)	'WALD'|'IGAUSS'
Weibull	'WEIBULL'

Note Except for T, F, and NORMALMIX, you can identify any distribution by its first four characters.

quantile

is a numeric constant, variable, or expression that specifies the value of the random variable.

Data type DOUBLE

parameter-1, ..., parameter-k
are optional numeric constants, variables, or expressions that specify the values of *shape*, *location*, or *scale* parameters that are appropriate for the specific distribution.

Data type DOUBLE

See Also

Functions:

- "CDF Function" on page 312
- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626
- "LOGSDF Function" on page 628
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

References

Shmueli, G., T. Minka,, S. Borle, and P. Boatwright. 2005. "A Useful Distribution for Fitting Discrete Data: Revival of the Conway-Maxwell-Poisson Distribution." *Applied Statistics* : 54:127–142.

PDF BERNOULLI Distribution Function

Returns a value from the Bernoulli probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('BERNOULLI', *x*, *p*)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

p

is a numeric constant, variable, or expression that specifies the probability of success.

Range $0 \le p \le 1$

Data type DOUBLE

Details

The PDF function for the Bernoulli distribution returns the probability density function with the probability of success equal to *p*. The PDF function is evaluated at the value *x*.

$$PDF('BERN', x, p) = \begin{cases} 0 & x < 0 \\ 1 - p & x = 0 \\ 0 & 0 < x < 1 \\ p & x = 1 \\ 0 & x > 1 \end{cases}$$

Note: There are no *location* or *scale* parameters for this distribution.

Example

The following statement illustrates the PDF Bernoulli distribution function:

Statements	Results
`select pdf('BERN', 0, .25);`	`0.75`

See Also

Functions:

- "CDF BERNOULLI Distribution Function" on page 315
- "LOGCDF Function" on page 622

- "LOGPDF Function" on page 626
- "LOGSDF Function" on page 628
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

PDF BETA Distribution Function

Returns a value from the beta probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('BETA', *x*, *a*, *b* [, *l*, *r*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $a > 0$

Data type DOUBLE

b

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $b > 0$

Data type DOUBLE

l

is a numeric constant, variable, or expression that specifies the left location parameter.

Default 0

Data type DOUBLE

r

is a numeric constant, variable, or expression that specifies the right location parameter.

Default 1

Range *r* > *l*

Data type DOUBLE

Details

The PDF function for the beta distribution returns the probability density function with the shape parameters *a* and *b*. The PDF function is evaluated at the value *x*.

$$PDF('BETA', x, a, b, l, r) = \begin{cases} 0 & x < l \\ \frac{1}{\beta(a,b)} \frac{(x-l)^{a-1}(r-x)^{b-1}}{(r-l)^{a+b-1}} & l \le x \le r \\ 0 & x > r \end{cases}$$

Note: The quantity $\frac{x-l}{r-l}$ is forced to be $\varepsilon \le \frac{x-l}{r-l} \le 1 - 2\varepsilon$.

Example

The following statement illustrates the PDF Beta distribution function:

Statements	Results
`select pdf('BETA', 0.2, 3, 4);`	1.2288

See Also

Functions:

- “CDF BETA Distribution Function” on page 316
- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “QUANTILE Function” on page 774

- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

PDF BINOMIAL Distribution Function

Returns a value from the binomial probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('BINOMIAL', *m*, *p*, *n*)

Arguments

m

is a random variable that counts the number of successes. This argument must be a whole number.

Range *m*=0, 1, ...

Data type DOUBLE

p

is a numeric constant, variable, or expression that specifies the probability of success.

Range $0 \le p \le 1$

Data type DOUBLE

n

is a parameter that counts the number of independent Bernoulli trials. This argument must be a whole number.

Range *n*=0, 1, ...

Data type DOUBLE

Details

The PDF function for the binomial distribution returns the probability density function with the parameters *p* and *n*. The PDF function is evaluated at the value *m*.

$$PDF('BINOM', m, p, n) = \begin{cases} 0 & m < 0 \\ \binom{n}{m} p^m (1-p)^{n-m} & 0 \le m \le n \\ 0 & m > n \end{cases}$$

Note: There are no *location* or *scale* parameters for the binomial distribution.

Example

The following statement illustrates the PDF Binomial distribution function:

Statements	Results
`select pdf('BINOM', 4, .5, 10);`	`0.205078`

See Also

Functions:

- "CDF BINOMIAL Distribution Function" on page 318
- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626
- "LOGSDF Function" on page 628
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

PDF CAUCHY Distribution Function

Returns a value from the Cauchy probability density (mass) distribution.

Categories: CAS
Probability

Alias: PMF

Returned data type: DOUBLE

Syntax

PDF ('CAUCHY', *x* [, *θ*, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The PDF function for the Cauchy distribution returns the probability density function with the location parameter θ and the scale parameter λ. The PDF function is evaluated at the value *x*.

$$PDF('CAUCHY', x, \theta, \lambda) = \frac{1}{\pi}\left(\frac{\lambda}{\lambda^2 + (x-\theta)^2}\right)$$

Example

The following statement illustrates the PDF Cauchy distribution function:

Statements	Results
`select pdf('CAUCHY', 2);`	`0.063662`

See Also

Functions:

- "CDF CAUCHY Distribution Function" on page 320
- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626
- "LOGSDF Function" on page 628
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

PDF Chi-Square Distribution Function

Returns a value from the chi-square probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('CHISQUARE', *x*, *df* [, *nc*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

df

is a numeric constant, variable, or expression that specifies the degrees of freedom.

Range *df* > 0

Data type DOUBLE

nc

is a numeric constant, variable, or expression that specifies an optional noncentrality parameter.

Range *nc* ≥ 0

Data type DOUBLE

Details

The PDF function for the chi-square distribution returns the probability density function of a chi-square distribution, with *df* degrees of freedom and the noncentrality parameter *nc*. The PDF function is evaluated at the value *x*. This function accepts noninteger degrees of freedom. If *nc* is omitted or equal to zero, the value returned is from the central chi-square distribution.

$$PDF('CHISQ', x, \nu, \lambda) = \begin{cases} 0 & x < 0 \\ \sum_{j=0}^{\infty} e^{-\frac{\lambda}{2}} \frac{\left(\frac{\lambda}{2}\right)^j}{j!} p_c(x, \nu + 2j) & x \geq 0 \end{cases}$$

In this equation, $p_c(.,.)$ denotes the density from the central chi-square distribution:

$$p_c(x, a) = \frac{1}{2} p_g\left(\frac{x}{2}, \frac{a}{2}\right)$$

In this equation, $p_g(y,b)$ is the density from the gamma distribution:

$$p_g(y, b) = \frac{1}{\Gamma(b)} e^{-y} y^{b-1}$$

Example

The following statement illustrates the PDF Chi-Square distribution function:

Statements	Results
`select pdf('CHISQ', 11.264, 11);`	`0.081686`

See Also

Functions:

- “CDF Chi-Square Distribution Function” on page 322
- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “QUANTILE Function” on page 774
- “SDF Function” on page 801

■ "SQUANTILE Function" on page 814

PDF Conway-Maxwell-Poisson Distribution Function

Returns a value from the Conway-Maxwell-Poisson probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF('CONMAXPOI',*y*,*λ*,*v*)

Arguments

y

is a numeric constant, variable, or expression that specifies a nonnegative, whole number representing a count.

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a location parameter, similar to the Poisson mean parameter.

Data type DOUBLE

v

is a numeric constant, variable, or expression that specifies a dispersion parameter.

Data type DOUBLE

Details

The Conway-Maxwell-Poisson (CMP) distribution is a generalization of the Poisson distribution that enables you to model underdispersed and overdispersed data. The CMP distribution is defined according to this equation:

$$P\left(Y = y\,;\lambda,\nu\right) = \frac{1}{Z(\lambda,\nu)}\frac{\lambda^{y}}{\left(y!\right)^{\nu}} \qquad y = 0, 1, 2, \ldots$$

The normalization factor is expressed by this equation:

$$Z\left(\lambda, \nu\right) = \sum_{n=0}^{\infty} \frac{\lambda^n}{(n!)^\nu}$$

λ and v are nonnegative and not simultaneously zero.

The additional parameter, v, allows for flexibility in modeling the tail behavior of the distribution. If v=1, the ratio is equal to the rate of decay of the Poisson distribution. If v<1, the rate of decay decreases, which enables you to model processes that have longer tails than the Poisson distribution (overdispersed data). If v>1, the rate of decay increases in a nonlinear manner, thus shortening the tail of the distribution (underdispersed data).

There are several special cases of the Conway-Maxwell-Poisson distribution. If λ<1 and v→∞, the Conway-Maxwell-Poisson distribution results in the Bernoulli distribution. In this case, the data can take only the values 0 and 1, which represent an extreme underdispersion. If v=1, the Poisson distribution is recovered with its equidispersion property. When v=0 and λ<1, the normalization factor is convergent and forms this geometric series:

$$Z\left(\lambda, 0\right) = \frac{1}{1-\lambda}$$

The probability density function is represented by this equation:

$$P\left(Y = y\,; \lambda, \nu = 0\right) = \left(1-\lambda\right)\lambda^y$$

The geometric distribution represents a case of severe overdispersion.

Mean, Variance, and Dispersion for the Conway-Maxwell-Poisson Model

The mean and variance of the Conway-Maxwell-Poisson distribution are defined by these equations:

$$E\left[Y\right] = \frac{\partial\, 1n\, Z}{\partial\, 1n\, \lambda}$$

$$V\left[Y\right] = \frac{\partial^2\, 1n\, Z}{\partial^2\, 1n\, \lambda}$$

The Conway-Maxwell-Poisson distribution does not have closed-form expressions for its moments in terms of parameters λ and v. However, the moments can be approximated. (For more information about the Conway-Maxwell-Poisson distribution and discrete data, see the References section that is located at the end of this function.) Use asymptotic expressions for Z to derive E(Y) and V(Y) as these equations show:

$$E\left[Y\right] \approx \lambda^{1/\nu} + \frac{1}{2\nu} - \frac{1}{2}$$

$$V\left[Y\right] \approx \frac{1}{\nu}\lambda^{1/\nu}$$

In the Conway-Maxwell-Poisson model, the summation of infinite series is evaluated using a logarithmic expansion. (For more information about the Conway-Maxwell-Poisson distribution and discrete data, see the References section that is located at the end of this function.) The mean and variance are calculated as follows for the Conway-Maxwell-Poisson model:

$$E\left(Y\right) = \frac{1}{Z(\lambda,\nu)} \sum_{j=0}^{\infty} \frac{j^{\lambda^j}}{(j!)^{\nu}}$$

$$V\left(Y\right) = \frac{1}{Z(\lambda,\nu)} \sum_{j=0}^{\infty} \frac{j^2\lambda^j}{(j!)^{\nu}} - E(Y)^2$$

The dispersion is defined as follows:

$$D\left(Y\right) = \frac{V(Y)}{E(Y)}$$

Example

The following statement illustrates the PDF Conway-Maxwell-Poisson distribution function:

Statements	Results
`select pdf('CONMAXPOI', .2, 2.3, .4);`	`0.009773`

See Also

Functions:

- “CDF Conway-Maxwell-Poisson Distribution Function” on page 324
- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

PDF EXPONENTIAL Distribution Function

Returns a value from the exponential probability density (mass) distribution.

Categories: CAS
Probability

Alias: PMF

Returned data type: DOUBLE

Syntax

PDF ('EXPONENTIAL', *x* [, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The PDF function for the exponential distribution returns the probability density function of an exponential distribution, with the scale parameter λ. The PDF function is evaluated at the value *x*.

$$PDF('EXPO', x, \lambda) = \begin{cases} 0 & x < 0 \\ \frac{1}{\lambda} exp\left(-\frac{x}{\lambda}\right) & x \geq 0 \end{cases}$$

Example

The following statement illustrates the PDF Exponential distribution function:

Statements	Results
`select pdf('EXPO', 1);`	`0.367879`

See Also

Functions:

- "CDF Exponential Distribution Function" on page 325
- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626
- "LOGSDF Function" on page 628
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

PDF F Distribution Function

Returns a value from the F probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('F', *x*, *ndf*, *ddf* [, *nc*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

ndf

is a numeric constant, variable, or expression that specifies the numerator degrees of freedom.

Range *ndf*> 0

Data type DOUBLE

ddf

is a numeric constant, variable, or expression that specifies the denominator degrees of freedom.

Range *ddf* > 0

Data type DOUBLE

nc

is a numeric constant, variable, or expression that specifies an optional noncentrality parameter.

Range *nc* ≥ 0

Data type DOUBLE

Details

The PDF function for the *F* distribution returns the probability density function of an *F* distribution, with *ndf* numerator degrees of freedom, *ddf* denominator degrees of freedom, and the noncentrality parameter *nc*. The PDF function is evaluated at the value *x*. This PDF function accepts noninteger degrees of freedom for *ndf* and *ddf*. If *nc* is omitted or equal to zero, the value returned is from a central *F* distribution. In the following equation, let $\nu_1 = ndf$, let $\nu_2 = ddf$, and let $\lambda = nc$. This equation describes the PDF function for the *F* distribution:

$$PDF('F', x, \nu_1, \nu_2, \lambda) = \begin{cases} 0 & x < 0 \\ \sum_{j=0}^{\infty} e^{-\frac{\lambda}{2}} \frac{\left(\frac{\lambda}{2}\right)^j}{j!} p_f(f, \nu_1 + 2j, \nu_2) & x \geq 0 \end{cases}$$

In the equation, $p_f(f,u_1,u_2)$ is the density from the central *F* distribution:

$$p_f(f, u_1, u_2) = p_B\left(\frac{u_1 f}{uf + u_2}, \frac{u_1}{2}, \frac{u_2}{2}\right) \frac{u_1 u_2}{(u_2 + u_1 f)^2}$$

In the equation, $p_B(x,a,b)$ is the density from the standard beta distribution.

Note: There are no *location* or *scale* parameters for the *F* distribution.

Example

The following statement illustrates the PDF F distribution function:

Statements	Results
`select pdf('F', 3.32, 2, 3);`	`0.054027`

See Also

Functions:

- "CDF F Distribution Function" on page 327
- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626
- "LOGSDF Function" on page 628
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

PDF GAMMA Distribution Function

Returns a value from the gamma probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('GAMMA', *x*, *a* [, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $a > 0$

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The PDF function for the gamma distribution returns the probability density function of a gamma distribution, with the shape parameter *a* and the scale parameter λ. The PDF function is evaluated at the value *x*.

$$PDF('GAMMA', x, a, \lambda) = \begin{cases} 0 & x < 0 \\ \frac{1}{\lambda^a \Gamma(a)} x^{a-1} exp\left(-\frac{x}{\lambda}\right) & x \geq 0 \end{cases}$$

Example

The following statement illustrates the PDF Gamma distribution function:

Statements	Results
`select pdf('GAMMA', 1, 3);`	`0.18394`

See Also

Functions:

- “CDF GAMMA Distribution Function” on page 329
- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “SDF Function” on page 801

PDF Generalized Poisson Distribution Function

Returns a value from the generalized Poisson probability density (mass) distribution.

Categories: CAS
Probability

Alias: PMF

Returned data type: DOUBLE

Syntax

PDF ('GENPOISSON', *x*, *θ*, *η*)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable. This argument must be a whole number.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $<10^5$ and >0

Data type DOUBLE

η

is a numeric constant, variable, or expression that specifies a shape parameter.

Range ≥0 and <0.95

Data type DOUBLE

Tip When η =0, the distribution is the Poisson distribution with a mean and variance of θ. When η>0, the mean is $\theta \div (1-\eta)$ and the variance is $\theta \div (1-\eta)^3$.

Details

Here is the probability mass function for the generalized Poisson distribution:

$$f(x;\theta,\eta) = \theta(\theta+\eta x)^{x-1}e^{-\theta-\eta x}/x!, \quad x = 0, 1, 2, ..., \qquad \theta > 0,\ 0 \le \eta < 1$$

Example

The following statement illustrates the PDF Generalized Poisson distribution function:

Statements	Results
`select pdf('GENPOISSON', 9, 1, .7);`	`0.015013`

See Also

Functions:

- “CDF Generalized Poisson Distribution Function” on page 330
- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

PDF GEOMETRIC Distribution Function

Returns a value from the geometric probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('GEOMETRIC', *m*, *p*)

Arguments

m

is a numeric constant, variable, or expression that specifies the number of failures before the first success.

Range $m \geq 0$

Data type DOUBLE

p

is a numeric constant, variable, or expression that specifies a probability of success.

Range $0 \leq p \leq 1$

Data type DOUBLE

Details

The PDF function for the geometric distribution returns the probability density function of a geometric distribution, with the parameter *p*. The PDF function is evaluated at the value *m*.

$$PDF('GEOM', m, p) = \begin{cases} 0 & m < 0 \\ p(1-p)^m & m \geq 0 \end{cases}$$

Note: There are no *location* or *scale* parameters for this distribution.

Example

The following statement illustrates the PDF Geometric distribution function:

Statements	Results
`select pdf('GEOMETRIC', 5, .3);`	`0.050421`

See Also

Functions:

- “CDF GEOMETRIC Distribution Function” on page 332
- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

PDF Hypergeometric Distribution Function

Returns a value from a hypergeometric probability density (mass) distribution.

Categories: CAS

	Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('HYPER', *x*, *N*, *R*, *n* [, *o*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable. This argument must be a whole number.

Data type DOUBLE

N

is a numeric constant, variable, or expression that specifies a population size parameter. This argument must be a whole number.

Range *N*=1, 2, ...

Data type DOUBLE

R

is a numeric constant, variable, or expression that specifies the number of items in the category of interest. This argument must be a whole number.

Range *R*=0, 1, ..., *N*

Data type DOUBLE

n

is a numeric constant, variable, or expression that specifies a sample size parameter. This argument must be a whole number.

Range *n*=1, 2, ..., *N*

Data type DOUBLE

o

is a numeric constant, variable, or expression that specifies an optional odds ratio parameter.

Range *o* > 0

Data type DOUBLE

Details

The PDF function for the hypergeometric distribution returns the probability density function of an extended hypergeometric distribution, with population size *N*, number of items *R*, sample size *n*, and odds ratio *o*. The PDF function is evaluated at the value *x*. If *o* is omitted or equal to 1, the value returned is from the usual hypergeometric distribution.

$$PDF\ ('HYPER', x, N, R, n, o) = \begin{cases} 0 & x < max(0, R+n-N) \\ \dfrac{\binom{R}{x}\binom{N-R}{n-x} o^x}{\sum_{j=max(0,R+n-N)}^{min(R,n)} \binom{R}{j}\binom{N-R}{n-j} o^j} & max(0, R+n-N) \le x \le min(R,n) \\ 0 & x > min(R,n) \end{cases}$$

Example

The following statement illustrates the PDF Hypergeometric distribution function:

Statements	Results
`select pdf('HYPER', 2, 200, 50, 10);`	`0.286851`

See Also

Functions:

- “CDF HYPERGEOMETRIC Distribution Function” on page 333
- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

PDF LAPLACE Distribution Function

Returns a value from the Laplace probability density (mass) distribution.

Categories: CAS

	Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('LAPLACE', *x* [, *θ*, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The PDF function for the Laplace distribution returns the probability density function of the Laplace distribution, with the location parameter θ and the scale parameter λ. The PDF function is evaluated at the value *x*.

$$PDF('LAPLACE', x, \theta, \lambda) = \frac{1}{2\lambda} exp\left(-\frac{|x-\theta|}{\lambda}\right)$$

Example

The following statement illustrates the PDF Laplace distribution function:

Statements	Results
`select pdf('LAPLACE', 1);`	`0.18394`

See Also

Functions:

- “CDF LAPLACE Distribution Function” on page 335
- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

PDF LOGISTIC Distribution Function

Returns a value from the logistic probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('LOGISTIC', *x* [, *θ*, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The PDF function for the logistic distribution returns the probability density function of a logistic distribution, with the location parameter θ and the scale parameter λ. The PDF function is evaluated at the value *x*.

$$PDF('LOGISTIC', x, \theta, \lambda) = \frac{exp\left(-\frac{x-\theta}{\lambda}\right)}{\lambda\left(1 + exp\left(-\frac{x-\theta}{\lambda}\right)\right)^2}$$

Example

The following statement illustrates the PDF Logistic distribution function:

Statements	Results
`select pdf('LOGISTIC', 1);`	`0.196612`

See Also

Functions:

- “CDF LOGISTIC Distribution Function” on page 337
- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

PDF LOGNORMAL Distribution Function

Returns a value from the lognormal probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('LOGNORMAL', *x* [, *θ*, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a log scale parameter. exp(θ) is a scale parameter.

Default 0

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a shape parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The PDF function for the lognormal distribution returns the probability density function of a lognormal distribution, with the log scale parameter θ and the shape parameter λ. The PDF function is evaluated at the value *x*.

$$PDF('LOGN', x, \theta, \lambda) = \begin{cases} 0 & x \leq 0 \\ \frac{1}{x\lambda\sqrt{2\pi}} \; exp\left(-\frac{(log(x)-\theta)^2}{2\lambda^2}\right) & x > 0 \end{cases}$$

Example

The following statement illustrates the PDF Lognormal distribution function:

Statements	Results
`select pdf('LOGNORMAL', 1);`	`0.398942`

See Also

Functions:

- “CDF LOGNORMAL Distribution Function” on page 339
- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

PDF NEGBINOMIAL Distribution Function

Returns the value from the negative binomial probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('NEGBINOMIAL', *m*, *p*, *n*)

Arguments

m

is a numeric constant, variable, or expression that specifies a random variable that counts the number of failures. This argument must be a positive, whole number.

Range *m*=0, 1, ...

Data type DOUBLE

p

is a numeric constant, variable, or expression that specifies a probability of success.

Range $0 \le p \le 1$

Data type DOUBLE

n

is a numeric constant, variable, or expression that specifies a value that counts the number of successes.

Range *n*>0

Data type DOUBLE

Details

The PDF function for the negative binomial distribution returns the probability density function of a negative binomial distribution, with probability of success *p* and number of successes *n*. The PDF function is evaluated at the value *m*.

$$PDF('NEGB', m, p, n) = \begin{cases} 0 & m < 0 \\ \binom{n+m-1}{n-1} p^n (1-p)^m & m \ge 0 \end{cases}$$

Note: There are no *location* or *scale* parameters for the negative binomial distribution.

Example

The following statement illustrates the PDF Negative Binomial distribution function:

Statements	Results
`select pdf('NEGB',1,.5,2);`	`0.25`

See Also

Functions:

- “CDF NEGBINOMIAL Distribution Function” on page 340
- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “SDF Function” on page 801

PDF NORMAL Distribution Function

Returns a value from the normal probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('NORMAL', *x* [, *θ*, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The PDF function for the normal distribution returns the probability density function of a normal distribution, with the location parameter θ and the scale parameter λ. The PDF function is evaluated at the value *x*.

$$PDF('NORMAL', x, \theta, \lambda) = \frac{1}{\lambda\sqrt{2\pi}}\ exp\left(-\frac{(x-\theta)^2}{2\lambda^2}\right)$$

Example

The following statement illustrates the PDF Normal distribution function:

Statements	Results
`select pdf('NORMAL', 1.96);`	`0.058441`

See Also

Functions:

- “CDF NORMAL Distribution Function” on page 342
- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

PDF NORMALMIX Distribution Function

Returns a value from the normal mixture probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('NORMALMIX', *x*, *n*, p_1, p_2, ... p_n, m_1, m_2, ... m_n, s_1, s_2, ... s_n)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

n

is a numeric constant, variable, or expression that specifies the number of mixtures. This argument must be a whole number.

Range *n*=1, 2, ...

Data type DOUBLE

p_i

is a list of numeric constants, variables, or expressions that specifies the *n* proportions, $p_1, p_2, \ldots, p_n$, where $\sum_{i=1}^{i=n} p_i = 1$.

Range *p*=0, 1, ...

Data type DOUBLE

m_i

is a list of numeric constants, variables, or expressions that specifies the *n* means $m_1, m_2, \ldots, m_n$.

s_i

is a list of numeric constants, variables, or expressions that specifies the *n* standard deviations $s_1, s_2, \ldots, s_n$.

Range *s* > 0

Data type DOUBLE

Details

The PDF function for the Normal Mixture distribution returns the probability that an observation from a mixture of normal distribution is less than or equal to *x*.

$$PDF('NORMALMIX', x, n, p, m, s) = \sum_{i=1}^{i=n} p_i \, PDF('NORMAL', x, m_i, s_i)$$

Weights for the Normal Mixture distribution must be nonnegative. If the sum of the weights does not equal 1, then the weights are treated as relative weights and adjusted so that the sum equals 1.

Note: There are no *location* or *scale* parameters for the Normal Mixture distribution.

Example

The following statement illustrates the PDF Normal Mixture distribution function:

Statements	Results
`select pdf('NORMALMIX',2.3, 3, .33, .33, .34,.5, 1.5, 2.5,.79, 1.6, 4.3);`	`0.116554`

See Also

Functions:

- “CDF NORMALMIX Distribution Function” on page 344
- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

PDF PARETO Distribution Function

Returns a value from the Pareto probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('PARETO', *x*, *a* [, *k*])

Arguments

x

is a numeric constant, variable, or expression that specifies a numeric random variable.

Data type DOUBLE

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $a > 0$

Data type DOUBLE

k

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range $k > 0$

Data type DOUBLE

Details

The PDF function for the Pareto distribution returns the probability density function of a Pareto distribution, with the shape parameter *a* and the scale parameter *k*. The PDF function is evaluated at the value *x*.

$$PDF('PARETO', x, a, k) = \begin{cases} 0 & x < k \\ \frac{a}{k}\left(\frac{k}{x}\right)^{a+1} & x \geq k \end{cases}$$

Example

The following statement illustrates the PDF Pareto distribution function:

Statements	Results
`select pdf('PARETO', 1, 1);`	1

See Also

Functions:

- "CDF PARETO Distribution Function" on page 346

- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626
- "LOGSDF Function" on page 628
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

PDF POISSON Distribution Function

Returns a value from the Poisson probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('POISSON', *n*, *m*)

Arguments

n

is a numeric constant, variable, or expression that specifies a random variable. This argument must be a whole number.

Range *n*=0, 1, ...

Data type DOUBLE

m

is a numeric constant, variable, or expression that specifies a mean parameter.

Range *m* > 0

Data type DOUBLE

Details

The PDF function for the Poisson distribution returns the probability density function of a Poisson distribution, with mean *m*. The PDF function is evaluated at the value *n*.

$$PDF('POISSON', n, m) = \begin{cases} 0 & n < 0 \\ e^{-m}\frac{m^n}{n!} & n \geq 0 \end{cases}$$

Note: There are no *location* or *scale* parameters for the Poisson distribution.

Example

The following statement illustrates the PDF Poisson distribution function:

Statements	Results
`select pdf('POISSON', 2, 1);`	`0.18394`

See Also

Functions:

- "CDF POISSON Distribution Function" on page 347
- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626
- "LOGSDF Function" on page 628
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

PDF T Distribution Function

Returns a value from the T probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('T', *t*, *df* [, *nc*])

Arguments

t

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

df

is a numeric constant, variable, or expression that specifies the degrees of freedom.

Range *df* > 0

Data type DOUBLE

nc

is a numeric constant, variable, or expression that specifies an optional noncentrality parameter.

Data type DOUBLE

Details

The PDF function for the *T* distribution returns the probability density function of a *T* distribution, with degrees of freedom *df* and the noncentrality parameter *nc*. The PDF function is evaluated at the value *x*. This PDF function accepts noninteger degrees of freedom. If *nc* is omitted or equal to zero, the value returned is from the central *T* distribution. In this equation, let $\nu = df$ and let $\delta = nc$.

$$PDF('T', t, \nu, \delta) = \frac{1}{2^{\frac{\nu}{2}-1}\Gamma\left(\frac{\nu}{2}\right)} \int_0^{\infty} x^{\nu-1} e^{-\frac{1}{2}x^2} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{tx}{\sqrt{\nu}}-\delta\right)^2} \frac{x}{\sqrt{\nu}} dx$$

Note: There are no *location* or *scale* parameters for the *T* distribution.

Example

The following statement illustrates the PDF T distribution function:

Statements	Results
`select pdf('T',.9,5);`	`0.24194`

See Also

Functions:

- "CDF T Distribution Function" on page 349
- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626
- "LOGSDF Function" on page 628
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

PDF TWEEDIE Distribution Function

Returns a value from the Tweedie probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('TWEEDIE', *y*, *p* [, *μ*, *φ*])

Arguments

y

is a numeric constant, variable, or expression that specifies a random variable.

Range	$y \geq 0$
Data type	DOUBLE
Notes	This argument is required. When $y>1$, y is numeric. When $p=1$, y is a whole number.

p

is a numeric constant, variable, or expression that specifies the power parameter.

Range	$p \geq 1$

Data type DOUBLE

Note This argument is required.

μ

is a numeric constant, variable, or expression that specifies the mean parameter.

Default 1

Range $\mu > 0$

Data type DOUBLE

φ

is a numeric constant, variable, or expression that specifies the dispersion parameter.

Default 1

Range $\varphi > 0$

Data type DOUBLE

Details

The PDF function for the Tweedie distribution returns an exponential dispersion model with variance and mean related by the equation *variance*= $\phi * \mu^p$.

$$\frac{1}{y}\sum_{j=1}^{\infty}\left(\frac{y^{-j\alpha}(p-1)^{j\alpha}}{\phi^{j(1-\alpha)}(2-p)^{j}j!\Gamma(-j\alpha)}\right)\exp\left(\frac{1}{\phi}\left(y\frac{\mu^{1-p}-1}{1-p}-\frac{\mu^{2-p}-1}{2-p}\right)\right)$$

The following relationship applies to the preceding equation:

$$\alpha = \frac{2-p}{1-p}$$

Note: The accuracy of computed Tweedie probabilities is highly dependent on the location in parameter space. Ten digits of accuracy are usually available except when p is near 2 or ϕ is near 0. In either of these cases, the accuracy might be as low as six digits.

Note: To avoid issues with numerical data, μ and Φ cannot be less than the constant SQRTMACEPS.

Example

The following statement illustrates the PDF Tweedie distribution function:

Statements	Results
`select pdf('TWEEDIE', .8, 5);`	`0.742291`

See Also

Functions:

- "CDF TWEEDIE Distribution Function" on page 351
- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626
- "LOGSDF Function" on page 628
- "QUANTILE Function" on page 774
- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

PDF UNIFORM Distribution Function

Returns a value from the uniform probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('UNIFORM', *x* [, *l*, *r*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

l

is a numeric constant, variable, or expression that specifies the left location parameter.

Default 0

Data type DOUBLE

r

is a numeric constant, variable, or expression that specifies the right location parameter.

Default 1

Range *r* > *l*

Data type DOUBLE

Details

The PDF function for the uniform distribution returns the probability density function of a uniform distribution, with the left location parameter *l* and the right location parameter *r*. The PDF function is evaluated at the value *x*.

$$PDF('UNIFORM', x, l, r) = \begin{cases} 0 & x < l \\ \frac{1}{r-l} & l \le x \le r \\ 0 & x > r \end{cases}$$

Example

The following statement illustrates the PDF Uniform distribution function:

Statements	Results
`select pdf('UNIFORM', 0.25);`	`1`

See Also

Functions:

- “CDF UNIFORM Distribution Function” on page 353
- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “QUANTILE Function” on page 774
- “SDF Function” on page 801

■ "SQUANTILE Function" on page 814

PDF Wald (Inverse Gaussian) Distribution Function

Returns a value from the Wald (also known as the inverse Gaussian) probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	Double

Syntax

PDF ('WALD', *x*, *λ* [, *μ*])
PDF ('IGAUSS', *x*, *λ* [, *μ*])

Arguments

x
is a numeric constant, variable, or expression that specifies a random variable.

λ
is a numeric constant, variable, or expression that specifies a shape parameter.

Range $\lambda > 0$

μ
is a numeric constant, variable, or expression that specifies the mean parameter.

Default 1

Range $\mu > 0$

Details

The PDF function for the Wald distribution returns the probability density function of a Wald distribution, with the shape parameter *λ*, which is evaluated at the value *x*.

$$fx(x) = \left[\frac{\lambda}{2\pi x^3}\right]^{1/2} \exp\left\{-\frac{\lambda}{2\mu^2 x}(x-\mu)^2\right\}, \qquad x > 0$$

Example

The following statement illustrates the PDF Wald distribution function:

Statements	Results
`select pdf('WALD', 1, 2);`	`0.56419`

See Also

Functions:

- “CDF WALD (Inverse Gaussian) Distribution Function” on page 354
- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “QUANTILE Function” on page 774
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

PDF WEIBULL Distribution Function

Returns a value from the Weibull probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF('WEIBULL', *x*, *a* [, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $a > 0$

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range $\lambda > 0$

Data type DOUBLE

Details

The PDF function for the Weibull distribution returns the probability density function of a Weibull distribution, with the shape parameter *a* and the scale parameter λ. The PDF function is evaluated at the value *x*.

$$PDF('WEIBULL', x, a, \lambda) = \begin{cases} 0 & x < 0 \\ exp\left(-\left(\frac{x}{\lambda}\right)^a\right)\frac{a}{\lambda}\left(\frac{x}{\lambda}\right)^{a-1} & x \geq 0 \end{cases}$$

Example

The following statement illustrates the PDF Weibull distribution function:

Statements	Results
`select pdf('WEIBULL', 1, 2);`	`0.735759`

See Also

Functions:

- "CDF WEIBULL Distribution Function" on page 356
- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626
- "LOGSDF Function" on page 628
- "QUANTILE Function" on page 774

- "SDF Function" on page 801
- "SQUANTILE Function" on page 814

PERM Function

Computes the number of permutations of *n* items that are taken *r* at a time.

Categories:	CAS Combinatorial
Returned data type:	DOUBLE

Syntax

PERM(*n*[, *r*])

Arguments

n

specifies any valid expression that represents the total number of elements from which the sample is chosen.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

r

Specifies any valid expression that represents the number of chosen elements.

Restriction $r \leq n$

Data type DOUBLE

Note If *r* is omitted, the function returns the factorial of *n*.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The mathematical representation of the PERM function is given by the following equation:

$$PERM(n, r) = \frac{n!}{(n-r)!}$$

with $n \geq 0$, $r \geq 0$, and $n \geq r$.

If the expression cannot be computed, a missing value is returned. For moderately large values, it is sometimes not possible to compute the PERM function.

Example

The following statements illustrate the PERM function:

Statements	Results
`select perm(5, 1);`	`5`
`select perm(5);`	`120`
`select perm(5, 2);`	`20`

See Also

Functions:

- "COMB Function" on page 367
- "FACT Function" on page 442

PI Function

Returns the constant value of PI as a floating-point value.

Categories:	Mathematical CAS
Returned data type:	DOUBLE

Syntax

PI()

Details

The value of PI is 3.141593.

Example

The following statement illustrates the PI function:

Statements	Results
`select pi();`	`3.141593`

PMT Function

Returns the periodic payment for a constant payment loan or the periodic savings for a future balance.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

PMT(*rate*, *number-of-periods*, *principal-amount*[, *future-amount*][, *type*])

Arguments

rate
specifies the interest rate per payment period.

Data type DOUBLE

number-of-periods
specifies the number of payment periods.

Requirement *Number-of-periods* must be a positive whole number.

Data type DOUBLE

principal-amount
specifies the principal amount of the loan. Zero is assumed if a null or missing value is specified.

Data type DOUBLE

future-amount

specifies the future amount. *Future-amount* can be the outstanding balance of a loan after the specified number of payment periods, or the future balance of periodic savings. Zero is assumed if *future-amount* is omitted or if a missing value is specified.

Data type DOUBLE

type

specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if *type* is omitted or if a null or missing value is specified.

Data type DOUBLE

Example

- The monthly payment for a $10,000 loan with a nominal annual interest rate of 8% and 10 end-of-month payments can be computed in the following ways:

  ```
  select PMT(0.08/12., 10, 10000, 0, 0);
  ```

  ```
  select PMT(0.08/12., 10, 10000);
  ```

  ```
  1037.03208935915
  ```

  These computations return a value of 1037.032.

- If the same loan has beginning-of-period payments, then payment can be computed as follows:

  ```
  select PMT(0.08/12., 10, 10000, 0, 1);
  ```

  ```
  1030.16432717796
  ```

  This computation returns a value of 1030.164.

- The payment for a $5,000 loan earning a 12% nominal annual interest rate that is to be paid back in five monthly payments is computed as follows:

  ```
  select PMT(.01/12., 5, 5000);
  ```

  ```
  1002.50138831008
  ```

  This computation returns a value of 1002.501.

- The payment for monthly periodic savings that accrue more than 18 years at a 6% nominal annual interest rate and that accumulate $50,000 at the end of the 18 years is computed as follows:

  ```
  select PMT(0.06/12., 216, 0, 50000, 0);
  ```

  ```
  -129.081160867993
  ```

  This computation returns a value of -129.081.

POISSON Function

Returns the probability from a Poisson distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE

Syntax

POISSON(*m*, *n*)

Arguments

m

specifies any valid expression that evaluates to a numeric mean parameter.

Range $m \geq 0$

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

n

specifies any valid expression that evaluates to a random variable.

Range $n \geq 0$

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

Details

The POISSON function returns the probability that an observation from a Poisson distribution, with mean *m*, is less than or equal to *n*. To compute the probability that an observation is equal to a given value, *n*, compute the difference of two probabilities from the Poisson distribution for *n* and $n-1$.

Example

The following statement illustrates the POISSON function:

Statements	Results
`select poisson(1, 2);`	`0.9196986029`

POWER Function

Returns the value of a numeric value expression raised to a specified power.

Categories: Mathematical
CAS

Returned data type: DOUBLE

Syntax

POWER(*numeric-expression*, *power-numeric-expression*)

Required Arguments

numeric-expression
specifies any valid SQL expression that evaluates to a numeric value.

Data type BIGINT, DOUBLE, FLOAT, INTEGER, REAL, SMALLINT, TINYINT

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

power-numeric-expression
specifies any valid SQL expression that evaluates to a numeric value. This argument is the power value.

Details

If *numeric-expression* is a null or missing value, then the POWER function returns a null or missing value. If the result is a number that does not fit into the range of a DOUBLE, the POWER function fails.

Example

Table: DENSITIES on page 1173

The following statements illustrate the POWER function:

Statements	Results
`select power(5*3, 2);`	`225`

`select name from densities`
` where squaremiles >= power(density,2);`
	`Afghanistan`
	`Algeria`
	`Angola`
	`Argentina`
	`Australia`

PROBBETA Function

Returns the probability from a beta distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBBETA(*x*, *a*, *b*)

Arguments

x

is a numeric random variable.

Range $0 \le x \le 1$

Data type DOUBLE

a

is a numeric shape parameter.

Range $a > 0$

Data type DOUBLE

b

is a numeric shape parameter.

Range $b > 0$

Data type DOUBLE

Details

The PROBBETA function returns the probability that an observation from a beta distribution, with shape parameters *a* and *b*, is less than or equal to *x*.

Example

The following statement illustrates the PROBBETA function:

Statements	Results
`select probbeta(.2,3,4);`	`0.09888`

PROBBNML Function

Returns the probability from a binomial distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBBNML(*p*, *n*, *m*)

Arguments

p
is a numeric probability of success parameter.

Range $0 \le p \le 1$

Data type DOUBLE

n

is the number of independent Bernoulli trials parameter. This argument must be a whole number.

Range $n > 0$

Data type DOUBLE

m

is the number of successes random variable. This argument must be a whole number.

Range $0 \le m \le n$

Data type DOUBLE

Details

The PROBBNML function returns the probability that an observation from a binomial distribution, with probability of success *p*, number of trials *n*, and number of successes *m*, is less than or equal to *m*. To compute the probability that an observation is equal to a given value *m*, compute the difference of two probabilities from the binomial distribution for *m* and *m*−1 successes.

Example

The following statement illustrates the PROBBNML function:

Statements	Results
`select probbnml(0.5,10,4);`	`0.376953125`

PROBBNRM Function

Returns a probability from a bivariate normal distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBBNRM(*x*, *y*, *r*)

Arguments

x

specifies a numeric constant, variable, or expression.

Data type DOUBLE

y

specifies a numeric constant, variable, or expression.

Data type DOUBLE

r

is a numeric correlation coefficient.

Range -1 ≤ *r* ≤ 1

Data type DOUBLE

Details

The PROBBNRM function returns the probability that an observation (X, Y) from a standardized bivariate normal distribution with mean 0, variance 1, and a correlation coefficient *r*, is less than or equal to (*x*, *y*). That is, it returns the probability that X≤*x* and Y≤*y*. The following equation describes the PROBBNRM function, where *u* and *v* represent the random variables *x* and *y*, respectively:

$$\mathrm{PROBBNRM}(x, y, r) = \frac{1}{2\pi\sqrt{1-r^2}} \int_{-\infty}^{x} \int_{-\infty}^{y} \exp\left[-\frac{u^2 - 2ruv + v^2}{2(1-r^2)}\right] dv\, du$$

Example

The following statement illustrates the PROBBNRM function:

Statements	Results
`select probbnrm(.4, -.3, .2);`	`0.27831833451902`

PROBCHI Function

Returns the probability from a chi-square distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE

Syntax

PROBCHI(*x*, *df*[, *nc*])

Arguments

x

is a numeric random variable.

Range $x \geq 0$

Data type DOUBLE

df

is a numeric degrees of freedom parameter.

Range $df > 0$

Data type DOUBLE

nc

is an optional numeric noncentrality parameter.

Range $nc \geq 0$

Data type DOUBLE

Details

The PROBCHI function returns the probability that an observation from a chi-square distribution, with degrees of freedom *df* and noncentrality parameter *nc*, is less than or equal to *x*. This function accepts a noninteger degrees of freedom parameter *df*. If the optional parameter *nc* is not specified or has the value 0, the value returned is from the central chi-square distribution.

Example

The following statement illustrates the PROBCHI function:

Statements	Results
`select probchi(11.264,11);`	`0.5785813293173`

PROBF Function

Returns the probability from an *F* distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBF(*x*, *ndf*, *ddf*[, *nc*])

Arguments

x
is a numeric random variable.

Range $x \geq 0$

Data type DOUBLE

ndf
is a numeric numerator degrees of freedom parameter.

Range $ndf > 0$

Data type DOUBLE

ddf
is a numeric denominator degrees of freedom parameter.

Range $ddf > 0$

Data type DOUBLE

nc
is an optional numeric noncentrality parameter.

Range *nc* ≥ 0

Data type DOUBLE

Details

The PROBF function returns the probability that an observation from an *F* distribution, with numerator degrees of freedom *ndf*, denominator degrees of freedom *ddf*, and noncentrality parameter *nc*, is less than or equal to *x*. The PROBF function accepts noninteger degrees of freedom parameters *ndf* and *ddf*. If the optional parameter *nc* is not specified or has the value 0, the value returned is from the central *F* distribution.

The significance level for an *F* test statistic is given by the following equation.

```
p=1-probf(x,ndf,ddf);
```

Example

The following statement illustrates the PROBF function:

Statements	Results
`select probf(3.32,2,3);`	`0.8263933602`

PROBGAM Function

Returns the probability from a gamma distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBGAM(*x*, *a*)

Arguments

x
is a numeric random variable.

Range $x \geq 0$

Data type DOUBLE

a

is a numeric shape parameter.

Data type DOUBLE

Details

The PROBGAM function returns the probability that an observation from a gamma distribution, with shape parameter *a*, is less than or equal to *x*.

Example

The following statement illustrates the PROBGAM function:

Statements	Results
`select probgam(1,3);`	`0.0803013971`

PROBHYPR Function

Returns the probability from a hypergeometric distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBHYPR(*N*, *K*, *n*, *x*[, *r*])

Arguments

N

is a population size parameter. This argument must be a whole number.

Range $N \geq 1$

Data type DOUBLE

K

is the number of items in the category of interest parameter. This argument must be a whole number.

Range $0 \le K \le N$

Data type DOUBLE

n

is the sample size parameter. This argument must be a whole number.

Range $0 \le n \le N$

Data type DOUBLE

x

is the random variable. This argument must be a whole number.

Range $max(0, K + n - N) \le x \le min(K,n)$

r

is a numeric odds ratio parameter.

Range $r \ge 0$

Data type DOUBLE

Details

The PROBHYPR function returns the probability that an observation from an extended hypergeometric distribution, with population size *N*, number of items *K*, sample size *n*, and odds ratio *r*, is less than or equal to *x*. If the optional parameter *r* is not specified or is set to 1, the value returned is from the usual hypergeometric distribution.

Example

The following statement illustrates the PROBHYPR function:

Statements	Results
`select probhypr(200,50,10,2);`	`0.5236734081`

PROBIT Function

Returns a quantile from the standard normal distribution.

Categories:	CAS Quantile
Returned data type:	DOUBLE

Syntax

PROBIT(*p*)

Arguments

p

is a numeric probability.

Range 0 < *p* < 1

Data type DOUBLE

Details

The PROBIT function returns the pth quantile from the standard normal distribution. The probability that an observation from the standard normal distribution is less than or equal to the returned quantile is *p*.

CAUTION

The result could be truncated to lie between -8.222 and 7.941.

Note: PROBIT is the inverse of the PROBNORM function.

Example

The following statements illustrate the PROBIT function:

Statements	Results
`select probit(.025);`	`-1.959963985`
`select probit(1.e-7);`	`-5.199337582`

PROBMC Function

Returns a probability or a quantile from various distributions for multiple comparisons of means.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBMC(*distribution*, *q*, *prob*, *df*, *nparms*[, *parameters*])

Arguments

distribution
is a character constant, variable, or expression that identifies the distribution. The following distributions are valid:

Distribution	Argument
Analysis of Means	`ANOM`
One-sided Dunnett	`DUNNETT1`
Two-sided Dunnett	`DUNNETT2`
Maximum Modulus	`MAXMOD`
Partitioned Range	`PARTRANGE`
Studentized Range	`RANGE`
Williams	`WILLIAMS`

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

q
is the quantile from the distribution.

Restriction Either *q* or *prob* can be specified, but not both.

Data type DOUBLE

prob
is the left probability from the distribution.

Restriction Either *prob* or *q* can be specified, but not both.

Data type DOUBLE

df
is the degrees of freedom.

Note: A missing value is interpreted as an infinite value.

nparms
is the number of treatments.

Data type DOUBLE

Note For DUNNETT1 and DUNNETT2, the control group is not counted.

parameters
is a set of *nparms* parameters that must be specified to handle the case of unequal sample sizes. The meaning of *parameters* depends on the value of *distribution*. If *parameters* is not specified, equal sample sizes are assumed, which is usually the case for a null hypothesis.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

Overview

The PROBMC function returns the probability or the quantile from various distributions with finite and infinite degrees of freedom for the variance estimate.

The *prob* argument is the probability that the random variable is less than *q*. Therefore, *p*-values can be computed as 1– *prob*. For example, to compute the critical value for a 5% significance level, set *prob*= 0.95. The precision of the computed probability is $O(10^{-8})$ (absolute error); the precision of computed quantile is $O(10^{-5})$.

Note: The studentized range is not computed for finite degrees of freedom and unequal sample sizes.

Note: Williams' test is computed only for equal sample sizes.

Formulas and Parameters

The equations listed here define expressions that are used in equations that relate the probability, *prob*, and the quantile, *q*, for different distributions and different situations within each distribution. For these equations, let *v* be the degrees of freedom, *df*.

$$d\mu_\nu(x) = \frac{\nu^{\frac{\nu}{2}}}{\Gamma\left(\frac{\nu}{2}\right)2^{\frac{\nu}{2}-1}} x^{\nu-1}\varepsilon^{-\frac{\nu x^2}{2}}dx$$

$$\phi(x) = \frac{1}{\sqrt{2\pi}}\varepsilon^{-\frac{x^2}{2}}$$

$$\Phi(x) = \int_{-\infty}^{x} \phi(u)du$$

Computing the Analysis of Means

Analysis of Means (ANOM) applies to data that is organized as *k* (Gaussian) samples, the ith sample being of size n_i. Let $I = \sqrt{-1}$. The distribution function [1, 2, 3, 4, 5] is the CDF for the maximum absolute of a *k*-dimensional multivariate T vector, with ν degrees of freedom, and an associated correlation matrix $\rho_{ij} = -\alpha_i\alpha_j$. This equation can be written as follows.

$$\begin{aligned} prob &= r(|t_1| < h, |t_2| < h, ..., |t_k| < h) \\ &= \int_0^\infty \left\{ \int_0^\infty \prod_{j=0}^{j=k} g(sh, y, \alpha_j)\phi(y)dy \right\} d\mu_\nu(s) \end{aligned}$$

The following relationship applies to the preceding equation:

$$g(sh, y, \alpha_j) = \Phi\left(\frac{sh - y\alpha_j}{\sqrt{1+\alpha_j^2}}\right) - \Phi\left(\frac{-sh - y\alpha_j}{\sqrt{1+\alpha_j^2}}\right)$$

In this equation, $\Gamma(.)$, $\phi(.)$, and $\Phi(.)$, are the gamma function, the density, and the CDF from the standard normal distribution, respectively.

For $\nu = \infty$, the distribution reduces to this equation.

$$r(|t_1| < h, |t_2| < h, ..., |t_k| < h) = \int_0^\infty \prod_{j=0}^{j=k} g(h, y, \alpha_j)\phi(y)dy$$

The following relationship applies to the preceding equation:

$$g(h, y, \alpha_j) = \Phi\left(\frac{h - y\alpha_j}{\sqrt{1+\alpha_j^2}}\right) - \Phi\left(\frac{-h - y\alpha_j}{\sqrt{1+\alpha_j^2}}\right)$$

For the balanced case, the distribution reduces to the following equation:

$$r(|t_1| < h, |t_2| < h, ..., |t_n| < h) = \int_0^\infty f(h, y, \rho)^n \phi(y)dy$$

The following relationships apply to the preceding equation:

$$f(h, y, \rho) = \Phi\left(\frac{h - y\sqrt{\rho}}{\sqrt{1+\rho}}\right) - \Phi\left(\frac{-h - y\sqrt{\rho}}{\sqrt{1+\rho}}\right)$$

$$\rho = \frac{1}{n-1}$$

Here is the syntax for this distribution:

x=*probmc*('*anom*', *q*, *p*, *nu*, *n*[, *alpha*$_1$, ..., *alpha*$_n$]);

Arguments

x
is a numeric value with the returned result.

q
is a numeric value that denotes the quantile.

p
is a numeric value that denotes the probability. One of *p* and *q* must be missing.

nu
is a numeric value that denotes the degrees of freedom.

n
is a numeric value that denotes the number of samples.

alpha*$_i$, *i* = 1, ...,*k
are optional numeric values denoting the alpha values from the first equation of this distribution. See "Computing the Analysis of Means" on page 756.

Many-One *t*-Statistics: Dunnett's One-Sided Test

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with finite degrees of freedom. The *parameters* are λ_1, ..., λ_k, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_0^\infty \int_{-\infty}^\infty \phi(y) \prod_{i=1}^k \Phi\left(\frac{\lambda_i y + qx}{\sqrt{1-\lambda_i^2}}\right) dy\, du_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with finite degrees of freedom. No *parameters* are passed $\left(\lambda = \sqrt{\frac{1}{2}}\right)$, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_0^\infty \int_{-\infty}^\infty \phi(y)[\Phi(y + \sqrt{2qx})]^k dy\, du_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with infinite degrees of freedom. The *parameters* are λ_1, ..., λ_k, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^\infty \phi(y) \prod_{i=1}^k \Phi\left(\frac{\lambda_i y + q}{\sqrt{1-\lambda_i^2}}\right) dy$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with infinite degrees of freedom. No *parameters* are passed $\left(\lambda = \sqrt{\frac{1}{2}}\right)$, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^{\infty} \phi(y)[\Phi(y+\sqrt{2q})]^k dy$$

Many-One *t*-Statistics: Dunnett's Two-sided Test

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with finite degrees of freedom. The *parameters* are $\lambda_1, ..., \lambda_k$, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_{0}^{\infty} \int_{-\infty}^{\infty} \phi(y) \prod_{i=1}^{k} \left[\Phi\left(\frac{\lambda_i y + qx}{\sqrt{1-\lambda_i^2}} \right) - \Phi\left(\frac{\lambda_i y - qx}{\sqrt{1-\lambda_i^2}} \right) \right] dy\, du_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with finite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_{0}^{\infty} \int_{-\infty}^{\infty} \phi(y)[\Phi(y+\sqrt{2qx}) - \Phi(y-\sqrt{2qx})]^k dy\, du_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with infinite degrees of freedom. The *parameters* are $\lambda_1, ..., \lambda_k$, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^{\infty} \phi(y) \prod_{i=1}^{k} \left[\Phi\left(\frac{\lambda_i y + q}{\sqrt{1-\lambda_i^2}} \right) - \Phi\left(\frac{\lambda_i y - q}{\sqrt{1-\lambda_i^2}} \right) \right] dy$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with infinite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^{\infty} \phi(y)[\Phi(y+\sqrt{2q}) - \Phi(y-\sqrt{2q})]^k dy$$

Computing the Partitioned Range

RANGE applies to the distribution of the studentized range for *n* group means. PARTRANGE applies to the distribution of the partitioned studentized range. Let the *n* groups be partitioned into *k* subsets of size $n_1 + ... + n_k = n$. Then the partitioned range is the maximum of the studentized ranges in the respective subsets. The studentization factor is the same in all cases.

$$prob = \int_{0}^{\infty} \prod_{i=1}^{i=k} \left(\int_{-\infty}^{\infty} k\phi(y)(\Phi(y) - \Phi(y-qx))^{k-1} dy \right)^{n_i} d\mu_\nu(x)$$

Here is the syntax for this distribution:

x=*probmc*('*partrange*', *q*, *p*,*nu*, *k*, n_1 ,...,n_k);

Arguments

x

is a numeric value with the returned result (either the probability or the quantile).

q

is a numeric value that denotes the quantile.

p
is a numeric value that denotes the probability. One of *p* and *q* must be missing.

nu
is a numeric value that denotes the degrees of freedom.

k
is a numeric value that denotes the number of groups.

n_i i=1,...,k
are optional numeric values that denote the *n* values from the equation in this distribution. See "Computing the Partitioned Range" on page 758.

The Studentized Range

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with finite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_0^{\infty} \int_{-\infty}^{\infty} k\phi(y)[\Phi(y) - \Phi(y - qx)]^{k-1} dy\, du_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with infinite degrees of freedom. The *parameters* are $\sigma_1, \ldots, \sigma_k$, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^{\infty} \sum_{j=1}^{k} \left\{ \prod_{i=1}^{k} \left[\Phi\left(\frac{y}{\sigma_i}\right) - \Phi\left(\frac{y-q}{\sigma_i}\right) \right] \right\} \phi\left(\frac{y}{\sigma_j}\right) \frac{1}{\sigma_j} dy$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with infinite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^{\infty} k\phi(y)[\Phi(y) - \Phi(y - q)]^{k-1} dy$$

The Studentized Maximum Modulus

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with finite degrees of freedom. The *parameters* are $\sigma_1, \ldots, \sigma_k$, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_0^{\infty} \prod_{i=1}^{k} \left[2\Phi\left(\frac{qx}{\sigma_i}\right) - 1 \right] d\mu_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with finite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_0^{\infty} [2\Phi(qx) - 1]^k d\mu_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with infinite degrees of freedom. The *parameters* are $\sigma_1, \ldots, \sigma_k$, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \prod_{i=1}^{k} \left[2\Phi\left(\frac{q}{\sigma_i}\right) - 1 \right]$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with infinite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = [2\Phi(q) - 1]^k$$

Williams' Test

PROBMC computes the probabilities or quantiles from the distribution defined in Williams (1971, 1972). See ""References"" in *SAS Functions and CALL Routines: Reference*.The need for the Williams' Test arises when you compare the dose treatment means with a control mean to determine the lowest effective dose of treatment.

Note: Williams' Test is computed only for equal sample sizes.

Let $X_1, X_2, ..., X_k$ be identical independent N(0,1) random variables. Let Y_k denote their average given by the following equation.

$$Y_k = \frac{X_1 + X_2 + ... + X_k}{k}$$

It is required to compute the distribution of the following value.

$$(Y_k - Z)/S$$

Arguments

Y_k

is as defined previously.

Z

is an N(0,1) independent random variable.

S

is such that ½vS^2 is a χ^2 variable with v degrees of freedom.

As described in Williams (1971), the full computation is extremely lengthy, and is carried out in three stages. See ""References"" in *SAS Functions and CALL Routines: Reference*.

1. Compute the distribution of Y_k. It is the fundamental (expensive) part of this operation and it can be used to find both the density and the probability of Y_k. Let U_i be defined in this equation.

$$U_i = \frac{X_1 + X_2 + ... + X_i}{i}, \ i = 1, 2, ..., k$$

You can write a recursive expression for the probability of $Y_k > d$. The value of *d* can be any real number.

$$\begin{aligned}\Pr(Y_k > d) &= \Pr(U_1 > d) \\ &+ \Pr(U_2 > d, U_1 < d) \\ &+ \Pr(U_3 > d, U_2 < d, U_1 < d) \\ &+ \ldots \\ &+ \Pr(U_k > d, U_{k-1} < d, \ldots, U_1 < d) \\ &= \Pr(Y_{k-1} > d) + \Pr(X_k + (k-1)U_{k-1} > kd)\end{aligned}$$

To compute this probability, start from an N(0,1) density function.

$$D(U_1 = x) = \phi(x)$$

And recursively compute the convolution.

$$D(U_k = x, U_{k-1} < d, \ldots, U_1 < d) =$$

$$\int_{-\infty}^{d} D(U_{k-1} = y, U_{k-2} < d, \ldots, U_1 < d)(k-1)\phi(kx - (k-1)y)dy$$

From this sequential convolution, it is possible to compute all the elements of the recursive equation for $\Pr(Y_k < d)$, shown previously.

2 Compute the distribution of $Y_k - Z$. This computation involves another convolution to compute the probability.

$$\Pr((Y_k - Z) > d) = \int_{-\infty}^{\infty} \Pr(Y_k > \sqrt{2d} + y)\phi(y)dy$$

3 Compute the distribution of $(Y_k - Z)/S$. This computation involves another convolution to compute the probability.

$$\Pr((Y_k - Z) > tS) = \int_{0}^{\infty} \Pr((Y_k - Z) > ty)d\mu_\nu(y)$$

The third stage is not needed when v = ∞. Due to the complexity of the operations, this lengthy algorithm is replaced by a much faster one when $k \le 15$ for both finite and infinite degrees of freedom v. For $k \ge 16$, the lengthy computation is carried out. It is extremely expensive and very slow due to the complexity of the algorithm.

Comparisons

The MEANS statement in the GLM Procedure of SAS/STAT Software computes the following tests:

- Dunnett's one-sided test
- Dunnett's two-sided test
- Studentized Range

Examples

Example 1: Computing the Analysis of Means

The following statements illustrate computing the analysis of means with the PROBMC function:

```
select probmc('anom',.,0.9,.,20) as q1,
       probmc('anom',.,0.9,20,5,0.1,0.1,0.1,0.1,0.1) as q2,
       probmc('anom',.,0.9,20,5,0.5,0.5,0.5,0.5,0.5) as q3,
       probmc('anom',.,0.9,20,5,0.1,0.2,0.3,0.4,0.5) as q4;
```

```
quit;
```

Output 4.11 *Results of Computing the Analysis of Means*

Q1	Q2	Q3	Q4
2.789506	2.454996	2.454996	2.453232

Example 2: Computing the Partitioned Range

The following statements illustrate computing the partitioned range with the PROBMC function:

```
select probmc('partrange',.,0.9,.,4,3,4,5,6) as q1,
       probmc('partrange',.,0.9,12,4,3,4,5,6) as q2;
quit;
```

Output 4.12 *Results of Computing the Partitioned Range*

Q1	Q2
4.10224	4.788863

Example 3: Computing Williams' Test

In the following program, a substance has been tested at seven levels in a randomized block design of eight blocks. The observed treatment means are as follows:

Treatment	Mean
X_0	10.4
X_1	9.9
X_2	10.0
X_3	10.6
X_4	11.4
X_5	11.9
X_6	11.7

The mean square, with (7 – 1)(8 – 1) = 42 degrees of freedom, is $s^2 = 1.16$.

Determine the maximum likelihood estimates M_i through the averaging process.

- Because $X_0 > X_1$, form $X_{0,1} = (X_0 + X_1)/2 = 10.15$.
- Because $X_{0,1} > X_2$, form $X_{0,1,2} = (X_0 + X_1 + X_2)/3 = (2X_{0,1} + X_2)/3 = 10.1$.
- $X_{0,1,2} < X_3 < X_4 < X_5$
- Because $X_5 > X_6$, form $X_{5,6} = (X_5 + X_6)/2 = 11.8$.

Now the order restriction is satisfied.

The maximum likelihood estimates under the alternative hypothesis are:

- $M_0 = M_1 = M_2 = X_{0,1,2} = 10.1$
- $M_3 = X_3 = 10.6$
- $M_4 = X_4 = 11.4$
- $M_5 = M_6 = X_{5,6} = 11.8$

Now compute $t = (11.8 - 10.4)/\sqrt{2s^2/8} = 2.60$, and the probability that corresponds to $k = 6$, v = 42, and $t = 2.60$ is .9924467341, which shows strong evidence that there is a response to the substance.

```
data _null_;
   dcl double prob quant5 quant1;
   method run();
      prob=probmc('WILLIAMS',2.6,.,42,6);
      put prob=;
      quant5=probmc('WILLIAMS',.,.95,42,6);
      put quant5=;
      quant1=probmc('WILLIAMS',.,.99,42,6);
      put quant1= ;
   end;
enddata;
run;
```

```
prob=0.99244668715827
quant5=1.80656253603894
quant1=2.49090827298707
```

You can also compute the quantiles for the upper 5% and 1% tails, as shown in the following table.

Statements	Results
`select probmc('williams',2.6,.,42,6) as prob;`	0.992447
`select probmc('williams',.,.95,42,6) as quant5;`	1.806563
`select probmc('williams',.,.99,42,6) as quant1;`	2.490908

References

Guirguis, G. H., and R. D. Tobias. 2004. “On the Computation of the Distribution for the Analysis of Means.” *Communications in Statistics: Simulation and Computation* 33: 861–887.

Nelson, P. R. 1981. “Numerical evaluation of an equicorrelated multivariate non-central t distribution.” *Communications in Statistics: Part B - Simulation and Computation* 10: 41–50.

Nelson, P.R. 1982a. “An Approximation for the Complex Normal Probability Integral.” *BIT* 22(1): 94–100.

Nelson, P. R. 1982b. “Exact critical points for the analysis of means.” *Communications in Statistics: Part A - Theory and Methods* 11: 699–709.

Nelson, P. R. 1988. “Application of the Analysis of Means.” *Proceedings of the SAS Users Group International Conference* 13: 225–230.

Nelson, P. R. 1991. “Numerical evaluation of multivariate normal integrals with correlations.” *The Frontiers of Statistical Scientific Theory and Industrial Applications* 2: 97–114.

Nelson, P. R. 1993. “Additional Uses for the Analysis of Means and Extended Tables of Critical Values.” *Technometrics* 35: 61–71.

PROBMED Function

Computes cumulative probabilities for the sample median.

Categories:	CAS Probability
Returned data type:	DOUBLE

Syntax

PROBMED(*n*, *x*)

Arguments

n

specifies the sample size.

Data type DOUBLE

x

is the point of interest. That is, the PROBMED function calculates the probability that the median is less than or equal to *x*.

Data type DOUBLE

Details

The PROBMED function computes the probability that the sample median is less than or equal to *x* for a sample of *n* independent, standard normal random variables (mean 0, variance 1).

Let *n* represent the sample size, and $x_{(i)}$ represents the *i*th order statistic. Then, when *n* is odd, the function makes the following calculation:

$$\Pr\left[X_{((n+1)/2)} \le x\right] = \mathrm{I}_{\Phi(x)}\left(\frac{n+1}{2}, \frac{n+1}{2}\right)$$

The following equations refer to the preceding equation:

$$\mathrm{I}_p(a,b) = \frac{\mathrm{I}}{\mathrm{B}(a,b)}\int_0^p t^{a-1}(\mathrm{I}-t)^{b-1}dt$$

In the equation $\mathrm{B}(a,b) = \Gamma(a)\Gamma(b)/\Gamma(a+b)$, Γ(.) is the gamma function. If *n* is even, the PROBMED function performs the following calculation:

$$\Pr[\frac{X_{(n/2)} + X_{((n/2)+1)}}{2} \le x] =$$

$$\frac{2}{\mathrm{B}(\frac{n}{2}, \frac{n}{2})}\int_{-\infty}^{x}\{[\mathrm{I}-\Phi(u)]^{n/2} - [\mathrm{I}-\Phi(2x-u)]^{n/2}\}[\Phi(u)]^{(n/2)-1}\phi(u)du$$

In this equation, $\mathrm{B}(n/2,\ n/2) = [\Gamma(n/2)]^2/\Gamma(n)$, and Φ(.) and φ(.) are the standard normal cumulative distribution function and density function, respectively.

Example

The following statement illustrates the PROBMED function:

Statements	Results
`select probmed(5,-0.1);`	`0.425638`

References

David, H.A. 1981. *Order Statistics*. 2nd ed. New York, New York: John Wiley & Sons.

PROBNEGB Function

Returns the probability from a negative binomial distribution.

Categories: CAS

Probability

Returned data type: DOUBLE

Syntax

PROBNEGB(*p*, *n*, *m*)

Arguments

p

is a numeric probability of success parameter.

Range $0 \le p \le 1$

Data type DOUBLE

n

is the number of successes parameter. This argument must be a whole number.

Range $n \ge 1$

Data type DOUBLE

m

is the random variable, the number of failures. This argument must be a positive, whole number.

Range $m \ge 0$

Data type DOUBLE

Details

The PROBNEGB function returns the probability that an observation from a negative binomial distribution, with probability of success *p* and number of successes *n*, is less than or equal to *m*.

To compute the probability that an observation is equal to a given value *m*, compute the difference of two probabilities from the negative binomial distribution for *m* and $m-1$.

Example

The following statement illustrates the PROBNEGB function:

Statements	Results
`select probnegb(0.5,2,1);`	`0.5`

PROBNORM Function

Returns the probability from the standard normal distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBNORM(*x*)

Arguments

x
is a numeric random variable.

Data type DOUBLE

Details

The PROBNORM function returns the probability that an observation from the standard normal distribution is less than or equal to *x*.

Note: PROBNORM is the inverse of the PROBIT function.

Example

The following statement illustrates the PROBNORM function:

Statements	Results
`select probnorm(1.96);`	`0.9750021049`

PROBT Function

Returns the probability from a t distribution of the values in an expression.

Categories:	Aggregate Descriptive Statistics CAS
Returned data type:	DOUBLE

Syntax

PROBT([DISTINCT] *expression*)

Arguments

expression
specifies any valid SQL expression.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

DISTINCT
(optional) eliminates duplicate rows from the aggregate calculation.

Details

The PROBT function returns the probability that an observation from a Student's t distribution, with degrees of freedom *n-1* and noncentrality parameter *nc* equal to 0, is less than or equal to *expression*.

The significance level of a two-tailed t test is given by this code line.

```
p=(1-probt(abs(x),df))*2;
```

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Comparisons

The STUDENTS_T function returns the Student's t-distribution. The PROBT function returns the probability that the Student's t-distribution is less than or equal to a given value.

Example

Table: DENSITIES on page 1173

The following statements illustrate the PROBT function:

Statements	Results
`select probt(density) from densities;`	`0.006068`
`select probt(population) from densities;`	`0.009722`

See Also

Functions:

- “STUDENTS_T Function” on page 822

SELECT Statement Clauses:

- “SELECT Clause” on page 956
- “GROUP BY Clause” on page 968
- “HAVING Clause” on page 970

PUT Function

Returns a value using a specified format.

Categories:	Special CAS
Returned data type:	NVARCHAR VARBINARY BINARY

Syntax

PUT(*source*, *format*.)

Arguments

source
identifies the variable or constant whose value you want to reformat.

Data type A data type that is supported by the format argument

format.
contains the SAS or FedSQL format that you want applied to the variable or constant that is specified in the source.

To override the default alignment, you can add an alignment specification to a format:

-L
left aligns the value

-C
centers the value

-R
right aligns the value

Restriction The format must be the same type as the value of *source*

Details

The PUT function affects the output of the query in which it is specified. It temporarily modifies the data type of the specified input variable.

The result of the PUT function is always a character string. If the source is numeric, the resulting string is right-aligned. If the source is character, the result is left-aligned.

Use PUT to format constants and to output stored data in a different format. Use PUT to convert a numeric value to a character value. For more information, see "How to Format Output with the PUT Function" on page 79.

Comparisons

The CAST function permanently modifies the data type of the specified input variable. The PUT function affects the output of the query in which it is specified.

Example

Table: WORLDTEMPS on page 1184

The following statement illustrates the PUT function.

Statement	Result
`select put(0.6666666667, fract8.);`	`2/3`
`select put(date(), date.);`	`19SEP13`

`select put(AvgLow, 4.1) from worldtemps;`	`45.0`
	`33.0`
	`17.0`
	`68.0`
	`56.0`
	`57.0`
	`28.0`
	`51.0`
	`75.0`
	`36.0`
	`33.0`
	`25.0`

See Also

"How to Format Output with the PUT Function" on page 79

PVP Function

Returns the present value for a periodic cash flow stream (such as a bond), with repayment of principal at maturity.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

PVP(A, c, n, K, k_0, y)

Arguments

A

specifies the par value.

Range $A > 0$

Data type DOUBLE

c

specifies the nominal per-year coupon rate, expressed as a fraction.

Range $0 \leq c < 1$

Data type DOUBLE

n

specifies the number of coupons per year.

Range $n > 0$

Data type DOUBLE

K

specifies the number of remaining coupons.

Range $K > 0$

Data type DOUBLE

k_0

specifies the time from the present date to the first coupon date, expressed in terms of the number of years.

Range $0 < k_0 \leq \frac{1}{n}$

Data type DOUBLE

y

specifies the nominal per-year yield-to-maturity, expressed as a fraction.

Range $y > 0$

Data type DOUBLE

Details

The PVP function is based on the following relationship:

$$P = \sum_{k=1}^{K} \frac{c(k)}{\left(1 + \frac{y}{n}\right)^{t_k}}$$

The following relationships apply to the preceding equation:

- $t_k = nk_0 + k - 1$
- $c(k) = \frac{c}{n}A \quad for\ k = 1, \ldots, K - 1$
- $c(K) = \left(1 + \frac{c}{n}\right)A$

Example

The following statement illustrates the PVP function.

Statements	Results
`select pvp(1000,.01,4,14,.33/2,.10);`	`743.1676`

QTR Function

Returns the quarter of the year from a SAS date value.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

QTR(*date*)

Arguments

date
specifies any valid expression that represents a SAS date value.

Data type DOUBLE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

Details

The QTR function returns a value of 1, 2, 3, or 4 from a SAS date value to indicate the quarter of the year in which a date value falls.

For more information about how FedSQL handles date and time values, see "Dates and Times in FedSQL" on page 58.

Example

The following statements illustrate the QTR function:

Statements	Results
`select qtr(16983);`	3
`select qtr(17075);`	4

See Also

Functions:

- "YYQ Function" on page 870

QUANTILE Function

Returns the quantile from a distribution when you specify the left probability (CDF).

Categories:	CAS Quantile
Returned data type:	Double
See:	"CDF Function" on page 312

Syntax

QUANTILE(*'distribution'*, *probability*, *parameter-1, …, parameter-k*)

Arguments

distribution

is a character constant, variable, or expression that identifies the distribution.

Note: The arguments for each of the QUANTILE distribution functions are identical to those of the corresponding CDF distribution functions.

Here are valid distributions:

Distribution	Argument
Bernoulli	'BERNOULLI'
Beta	'BETA'
Binomial	'BINOMIAL'
Cauchy	'CAUCHY'
Chi-Square	'CHISQUARE'
Conway-Maxwell-Poisson	'CONMAXPOI'
Exponential	'EXPONENTIAL'
F	'F'
Gamma	'GAMMA'
Generalized Poisson	'GENPOISSON'
Geometric	'GEOMETRIC'
Hypergeometric	'HYPERGEOMETRIC'
Laplace	'LAPLACE'
Logistic	'LOGISTIC'
Lognormal	'LOGNORMAL'
Negative binomial	'NEGBINOMIAL'
Normal	'NORMAL'|'GAUSS'
Normal mixture	'NORMALMIX'
Pareto	'PARETO'

Distribution	Argument
Poisson	'POISSON'
T	'T'
Tweedie	'TWEEDIE'
Uniform	'UNIFORM'
Wald (inverse Gaussian)	'WALD'|'IGAUSS'
Weibull	'WEIBULL'

Note: Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

probability
is a numeric constant, variable, or expression that specifies the value of a random variable.

parameter-1, …, parameter-k
are optional *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

Details

The QUANTILE function computes the quantile from the specified continuous or discrete distribution, based on the probability value that is provided. For more information, see the individual distributions noted in the table above.

The Conway-Maxwell-Poisson distribution for the QUANTILE function returns the counts value *y* that is the largest whole number whose CDF value is less than or equal to *p*. The syntax for the Conway-Maxwell-Poisson distribution in the QUANTILE function has the following form:

QUANTILE('CONMAXPOI',*p*,*λ*,*v*)

p
is a real number between 0 and 1, inclusively.

λ
is similar to the mean, as in the Poisson distribution.

v
is a dispersion parameter.

For more information, see “Conway-Maxwell-Poisson” distribution in the PDF function on page 704.

Example

The following statements illustrate the QUANTILE function:

Statements	Results
`select quantile('BERN', .75, .25);`	0
`select quantile('BETA', 0.1, 3, 4);`	0.200909
`select quantile('BINOM', .4, .5, 10);`	5
`select quantile('CAUCHY', .85);`	1.962611
`select quantile('CHISQ', .6, 11);`	11.52983
`select quantile('CONMAXPOI', .2, 2.3, .4);`	5
`select quantile('EXPO', .6);`	0.916291
`select quantile('F', .8, 2, 3);`	2.886027
`select quantile('GAMMA', .4, 3);`	2.285077
`select quantile('GENPOISSON', .9, 1, .7);`	9
`select quantile('HYPER', .5, 200, 50, 10);`	2
`select quantile('LAPLACE', .8);`	0.916291
`select quantile('LOGISTIC', .7);`	0.847298
`select quantile('LOGNORMAL', .5);`	1
`select quantile('NEGB', .5, .5, 2);`	1
`select quantile('NORMAL', .975);`	1.959964
`select quantile('NORMALMIX', 0.5, 1, 0.2, 1.1,0.1);`	1.1
`select quantile('PARETO', .01, 1);`	1.010101
`select quantile('POISSON', .9, 1);`	2
`select quantile('T', .8, 5);`	0.919544
`select quantile('TWEEDIE', .8, 5);`	1.26112
`select quantile('UNIFORM', 0.25);`	0.25
`select quantile('WALD', .6, 2);`	0.952621

Statements	Results
`select quantile('WEIBULL', .6, 2);`	`0.957231`

See Also

Functions:

- “CDF Function” on page 312
- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “PDF Function” on page 693
- “SDF Function” on page 801
- “SQUANTILE Function” on page 814

QUOTE Function

Adds double quotation marks to a character value.

Categories:	CAS Character
Returned data type:	CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

QUOTE(*expression*)

Arguments

expression

specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

Details

The QUOTE function adds double quotation marks, the default character, to a character value. If double quotation marks are found within the argument, they are doubled in the output.

The length of the receiving variable must be long enough to contain the argument (including trailing blanks), leading and trailing quotation marks, and any embedded quotation marks that are doubled. For example, if the argument is ABC followed by three trailing blanks, then the receiving variable must have a length of at least eight to hold “ABC###”. (The character # represents a blank space.) If the receiving field is not long enough, the QUOTE function returns a blank string, and writes an invalid argument note to the SAS log.

A string of characters enclosed in double quotation marks is a DS2 identifier and not a character constant. The double quotation marks become part of the identifier. Quoted identifiers cannot be used to create column names in an output table.

Example

The following statements illustrate the QUOTE function:

Statements	Results
`select quote('A"B');`	`"A""B"`
`select quote('A''B');`	`"A'B"`
`select quote('Paul''s Catering Service');`	`"Paul's Catering Service"`

RADIANS Function

Returns the number of radians converted from a numeric degree value.

Categories: Trigonometric
CAS

Returned data type: DOUBLE

Syntax

RADIANS(*expression*)

Arguments

expression
specifies any valid SQL expression that evaluates to a numeric value.

Data type BIGINT, DOUBLE, FLOAT, INTEGER, REAL, SMALLINT, TINYINT

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Example

The following statement illustrate the RADIANS function:

Statements	Results
`select radians(360);`	`6.283185`

See Also

Functions:

- "DEGREES Function" on page 421

RANGE Function

Returns the range between values in an expression.

Categories: Aggregate
Descriptive Statistics
CAS

Returned data type: the same type as the expression

Syntax

RANGE([DISTINCT] *expression*)

Arguments

expression
specifies any valid SQL expression.

Data type BIGINT, DOUBLE, FLOAT, INTEGER, REAL, SMALLINT, TINYINT

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

DISTINCT
(optional) eliminates duplicate rows from the aggregate calculation.

Details

The RANGE function returns the difference between the largest and the smallest values in the specified expression. The RANGE function ignores null values and SAS missing values.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Example

Table: WORLDTEMPS on page 1184

The following statement illustrates the RANGE function:

Statements	Results
`select range(AvgHigh) from worldtemps;`	27

See Also

SELECT Statement Clauses:

- "SELECT Clause" on page 956
- "GROUP BY Clause" on page 968
- "HAVING Clause" on page 970

RANK Function

Returns the position of a character in the ASCII collating sequence.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

RANK(*expression*)

Arguments

expression

specifies any valid expression that evaluates or can be coerced to a character string.

Data type	CHAR, NCHAR, NVARCHAR, VARCHAR
See	“<sql-expression> Expression” on page 894 “FedSQL Expressions” on page 46

Details

The RANK function returns a whole number that represents the position of the character in the ASCII collating sequence. When more than one character is specified, the RANK function returns the position in the ASCII collating sequence for the first character.

Note: Any program that uses the RANK function with characters above ASCII 127 is not portable. (The hexadecimal notation is `'7F'x`.) The program is not portable because these characters are national characters and they vary from country to country.

Example

The following statement illustrates the RANK function:

Statement	Result	
	ASCII	EBCDIC
`select "rank"('A');`	65	193

Note: RANK is a FedSQL reserved word. The RANK function must be specified as a delimited identifier.

See Also

Functions:

- "BYTE Function" on page 309

REPEAT Function

Repeats a character expression.

Categories:	CAS Character
Returned data type:	CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

REPEAT(*expression*, *n*)

Arguments

expression

specifies any valid expression that evaluates or can be coerced to a character string.

Data type	CHAR, NCHAR, NVARCHAR, VARCHAR
See	"<sql-expression> Expression" on page 894 "FedSQL Expressions" on page 46

n

specifies the number of times to repeat *expression*.

Restriction *n* must be greater than or equal to 0.

Data type BIGINT, DOUBLE

Details

The REPEAT function returns a character value consisting of the first argument repeated *n* times. Thus, the first argument appears *n*+1 times in the result.

Example

The following statement illustrates the REPEAT function:

Statements	Results
`select repeat('ONE',2);`	`ONEONEONE`

REVERSE Function

Reverses a character expression.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

REVERSE(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

Details

The REVERSE function returns a character value with the last character in the expression is the first character in the result, the next-to-last character in the expression is the second character in the result, and so on.

Note: Trailing blanks in the expression become leading blanks in the result.

Example

The following statement illustrates the REVERSE function:

Statements	Results
	`----+----1`
`select reverse('xyz  ');`	`zyx`

RMS Function

Returns the root mean square.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

RMS(*expression* [, …*expression*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The root mean square is the square root of the arithmetic mean of the squares of the values. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the root mean square of the non-null or nonmissing values.

Let n be the number of arguments with non-null or nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The root mean square is calculated as follows.

$$\sqrt{\frac{x_1^2 + x_2^2 + \ldots + x_n^2}{n}}$$

Example

The following statements illustrate the RMS function:

Statements	Results
`select rms(1,7);`	`5`
`select rms(.,1,5,11);`	`7`

ROUND Function

Rounds the first argument to the nearest multiple of the second argument, or to the nearest integer when the second argument is omitted.

Categories: CAS
Truncation

Returned data type: DOUBLE

Syntax

ROUND(*expression* [, *rounding-unit*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value, to be rounded.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

rounding-unit

specifies a positive numeric expression that specifies the rounding unit.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

Basic Concepts

The ROUND function rounds the first argument to a value that is very close to a multiple of the second argument. The results might not be an exact multiple of the second argument.

Differences between Binary and Decimal Arithmetic

Computers use binary arithmetic with finite precision. If you work with numbers that do not have an exact binary representation, computers often produce results that differ slightly from the results that are produced with decimal arithmetic.

For example, the decimal values 0.1 and 0.3 do not have exact binary representations. In decimal arithmetic, 3*0.1 is exactly equal to 0.3, but this equality is not true in binary arithmetic.

The Effects of Rounding

Rounding by definition finds an exact multiple of the rounding unit that is closest to the value to be rounded. For example, 0.33 rounded to the nearest tenth equals 3*0.1 or 0.3 in decimal arithmetic. In binary arithmetic, 0.33 rounded to the nearest tenth equals 3*0.1, and not 0.3, because 0.3 is not an exact multiple of one tenth in binary arithmetic.

The ROUND function returns the value that is based on decimal arithmetic, even though this value is sometimes not the exact, mathematically correct result. In the example `ROUND(0.33,0.1)`, ROUND returns 0.3 and not 3*0.1.

Testing for Approximate Equality

You should not use the ROUND function as a general method to test for approximate equality. Two numbers that differ only in the least significant bit can round to different values if one number rounds down and the other number rounds up. Testing for approximate equality depends on how the numbers have been

computed. If both numbers are computed to high relative precision, you could test for approximate equality by using the ABS and the MAX functions.

Producing Expected Results

In general, `ROUND(expression, rounding-unit)` produces the result that you expect from decimal arithmetic if the result has no more than nine significant digits and any of the following conditions are true:

- The rounding unit is an integer.
- The rounding unit is a power of 10 greater than or equal to 1e-15. [1]
- The result that you expect from decimal arithmetic has no more than four decimal places.

For example:

```
select round(1234.56789,100);          Result: 1200
select round(1234.56789,10);           Result: 1230
select round(1234.56789,1);            Result: 1235
select round(1234.56789,.1);           Result: 1234.6
select round(1234.56789,.01);          Result: 1234.57
select round(1234.56789,.001);         Result: 1234.568
select round(1234.56789,.0001);        Result: 1234.5679
select round(1234.56789,.00001);       Result: 1234.56789
select round(1234.56789,.1111);        Result: 1234.5432
select round(1234.56789,.11111);       Result: 1234.54321
```

When the Rounding Unit Is the Reciprocal of an Integer

When the rounding unit is the reciprocal of an integer [2], the ROUND function computes the result by dividing by the integer. Therefore, you can safely compare the result from ROUND with the ratio of two integers, but not with a multiple of the rounding unit.

Computing Results in Special Cases

The ROUND function computes the result by multiplying an integer by the rounding unit when all of the following conditions are true:

- The rounding unit is not an integer.
- The rounding unit is not a power of 10.
- The rounding unit is not the reciprocal of an integer.
- The result that you expect from decimal arithmetic has no more than four decimal places.

For example:

```
select round(1234.56789,.11111) - 11111*.11111;
```

1. If the rounding unit is less than one, ROUND treats it as a power of 10 if the reciprocal of the rounding unit differs from a power of 10 in at most the three or four least significant bits.
2. ROUND treats the rounding unit as a reciprocal of an integer if the reciprocal of the rounding unit differs from an integer in at most the three or four least significant bits.

Returns the value 0 (zero).

Computing Results When the Value Is Halfway between Multiples of the Rounding Unit

When the value to be rounded is approximately halfway between two multiples of the rounding unit, the ROUND function rounds up the absolute value and restores the original sign.

Example

The following statement illustrates the ROUND function:

Statement	Result
`select round(9.5,10);`	`10`

See Also

Functions:

- "CEIL Function" on page 358
- "CEILZ Function" on page 359
- "FLOOR Function" on page 520
- "FLOORZ Function" on page 521

ROUNDZ Function

Rounds the first argument to the nearest multiple of the second argument, using zero fuzzing.

Categories:	CAS Truncation
Returned data type:	DOUBLE

Syntax

ROUNDZ(*expression* [, *rounding-unit*])

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

rounding-unit

specifies any valid expression that evaluates to a numeric expression and that specifies the rounding unit.

Default 1

Requirement Only positive values are valid.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The ROUNDZ function rounds the first argument to the nearest multiple of the second argument.

Comparisons

The ROUNDZ function is the same as the ROUND function with these exceptions:

- ROUNDZ returns an even multiple when the first argument is exactly halfway between the two nearest multiples of the second argument. ROUND returns the multiple with the larger absolute value when the first argument is approximately halfway between the two nearest multiples.
- When the rounding unit is less than one and not the reciprocal of an integer, the result that is returned by ROUNDZ might not agree exactly with the result from decimal arithmetic. ROUNDZ does not fuzz the result. ROUND performs extra computations, called fuzzing, to try to make the result agree with decimal arithmetic.

Example: Sample Output from the ROUNDZ Function

The following statements illustrate the ROUNDZ function:

Statement	Results
`select put(roundz(223.456, 1), 9.5);`	223.00000
`select put(roundz(223.456, .01), 9.5);`	223.46000
`select put(roundz(223.456, 100), 9.5);`	200.00000
`select put(roundz(223.456, .3), 9.5);`	223.50000

See Also

Functions:

- "ROUND Function" on page 786

SAVING Function

Returns the future value of a periodic saving.

Category:	Financial
Restriction:	This function is not supported on the CAS server.
Returned data type:	DOUBLE

Syntax

SAVING(*f*, *p*, *r*, *n*)

Arguments

f

is numeric, the future amount (at the end of *n* periods).

Range $f \geq 0$

Data type DOUBLE

p

is numeric, the fixed periodic payment.

Range $p \geq 0$

Data type DOUBLE

r

is numeric, the periodic interest rate expressed as a decimal.

Range $r \geq 0$

Data type DOUBLE

n

is an integer, the number of compounding periods.

Range $n \geq 0$

Data type DOUBLE

Details

The SAVING function returns the missing argument in the list of four arguments from a periodic saving. The arguments are related by the following equation:

$$f = \frac{p(1+r)\left((1+r)^n - 1\right)}{r}$$

One missing argument must be provided. It is then calculated from the remaining three. No adjustment is made to convert the results to round numbers.

Example

A savings account pays a 5% nominal annual interest rate, compounded monthly. For a monthly deposit of $100, the number of payments that are needed to accumulate at least $12,000, can be expressed as follows:

```
select saving(12000, 100, .05/12, .);
```

The value that is returned is 97.18 months. The fourth argument is set to missing, which indicates that the number of payments is to be calculated. The 5% nominal annual rate is converted to a monthly rate of 0.05/12. The rate is the fractional (not the percentage) interest rate per compounding period.

See Also

Functions:

- “SAVINGS Function” on page 792

SAVINGS Function

Returns the balance of a periodic savings by using variable interest rates.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

SAVINGS(*base-date*, *initial-deposit-date*, *deposit-amount*, *deposit-number*, *deposit-interval*, *compounding-interval*, *date-1*, *rate-1*[, …*date-n*, *rate-n*])

Arguments

base-date
specifies the value that is returned is the balance of the savings at the base date.

Requirement *Base-date* is a SAS date.

Data type DOUBLE

initial-deposit-date
specifies the date of the first deposit. Subsequent deposits are at the beginning of subsequent deposit intervals.

Requirement *Initial-deposit-date* is a SAS date.

Data type DOUBLE

deposit-amount
specifies the value of each deposit. All deposits are assumed constant.

Data type DOUBLE

deposit-number
specifies the number of deposits.

Data type DOUBLE

deposit-interval
specifies the frequency at which deposits are made.

Requirement *Deposit-interval* is a SAS interval.

Data type CHAR

compounding-interval
specifies the compounding interval.

Requirement *Compounding-interval* is a SAS interval.

Data type CHAR

date
specifies the time at which *rate* takes effect. Each date is paired with a rate.

Requirement *Date* is a SAS date.

Data type DOUBLE

rate

specifies the interest rate as numeric percentage that starts on *date*. Each rate is paired with a date.

Data type DOUBLE

Details

The following details apply to the SAVINGS function:

- The values for rates must be between –99 and 120.
- *Deposit-interval* cannot be 'CONTINUOUS'.
- The list of date-rate pairs does not need to be in chronological order.
- When multiple rate changes occur on a single date, the SAVINGS function applies only the final rate that is listed for that date.
- Simple interest is applied for partial periods.
- There must be a valid date-rate pair whose date is at or prior to both the *initial-deposit-date* and the *base-date*.

Example

- If you deposit $300 monthly for two years into an account that compounds quarterly at an annual rate of 4%, the balance of the account after five years can be expressed as follows:

  ```
  select savings(date'2005-01-01', date'2000-01-01',
     300, 24, 'month', 'qtr', date'2000-01-01',4.00);
  ```

  The following line is written to the SAS log.

  ```
  8458.794159
  ```

- If the interest rate increases by a quarter-point each year, then the balance of the account could be expressed as follows:

  ```
  select savings(date'2005-01-01', date'2000-01-01',
     300, 24, 'month', 'qtr', date'2000-01-01', 4.00, date'2001-01-01', 4.25,
   date'2002-01-01', 4.50, date'2003-01-01', 4.75, date'2004-01-01', 5.0);
  ```

  The following line is written to the SAS log.

  ```
  8665.5059376
  ```

- To determine the balance after one year of deposits, the following program sets amount_base3 to the desired balance:

  ```
  select savings(date'2001-01-01', date'2000-01-01',
     300, 24, 'month', 'qtr', date'2000-01-01',4);
  ```

  The following line is written to the SAS log.

```
3978.69037121739
```

The SAVINGS function ignores deposits after the base date, so the deposits after the reference date do not affect the value that is returned.

See Also

Functions:

- "SAVING Function" on page 791

SCAN Function

Returns the *n*th word from a character expression.

Categories:	CAS Character
Returned data type:	CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

SCAN(*expression*, *n* [, *delimiters*[, *modifier*]])

Arguments

expression

specifies any valid expression that evaluates or can be coerced to a character string.

Data type	CHAR, NCHAR, NVARCHAR, VARCHAR
See	"<sql-expression> Expression" on page 894 "FedSQL Expressions" on page 46

n

is a nonzero numeric expression that specifies the number of the word in the character expression that you want SCAN to select. The following rules apply:

- If *n* is positive, SCAN counts words from left to right in the character string.
- If *n* is negative, SCAN counts words from right to left in the character string.
- If *n* is greater than the number of words in *expression*, SCAN returns a blank value.

delimiters

specifies any valid expression that evaluates or can be coerced to a character string and that SCAN uses as word separators in the expression.

Default

Requirement If *delimiter* is a constant, enclose *delimiter* in single quotation marks.

Interactions ASCII default delimiters are: blank ! $ % & ( ) * + , – . / ; < |. In environments without the ^ character, SCAN uses the ~ character instead.

EBCDIC default delimiters are: blank ! $ % & ( ) * + , – . / ; < ¬ | ¢.

Specifying a modifier can change the characters in *delimiter*. For example, if you specify the K modifier in the *modifier* argument, all characters that are not in *delimiter* are used as delimiters.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip You can add more characters to *delimiter* by using other modifiers.

See "Using Default Delimiters in ASCII and EBCDIC Environments" on page 798

"<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

modifier

specifies a character constant, variable, or expression in which each non-blank character modifies the action of the SCAN function. Blanks are ignored. Use the following characters as modifiers:

a or A adds alphabetic characters to the list of characters.

b or B scans backward from right to left instead of from left to right, regardless of the sign of the *count* argument.

c or C adds control characters to the list of characters.

d or D adds digits to the list of characters.

f or F adds an underscore and English letters to the list of characters.

g or G adds graphic characters to the list of characters. Graphic characters are characters that, when printed, produce an image on paper.

h or H adds a horizontal tab to the list of characters.

i or I ignores the case of the characters.

k or K causes all characters that are not in the list of characters to be treated as delimiters. That is, if K is specified, characters that are in the list of characters are kept in the returned value rather than being omitted because they are delimiters. If K is not specified, then all characters that are in the list of characters are treated as delimiters.

l or L adds lowercase letters to the list of characters.

m or M specifies that multiple consecutive delimiters, and delimiters at the beginning or end of the *string* argument, refer to words that have a length of zero. If the M modifier is not specified, then multiple consecutive delimiters are treated as one delimiter, and delimiters at the beginning or end of the *string* argument are ignored.

n or N adds digits, an underscore, and English letters to the list of characters.

o or O processes the *charlist* and *modifier* arguments only once, rather than every time the SCAN function is called. Using the O modifier in the DATA step (excluding WHERE clauses), or in the SQL procedure can make SCAN run faster when you call it in a loop where the *character-list* and *modifier* arguments do not change. The O modifier applies separately to each instance of the SCAN function in your SAS code, and does *not* cause all instances of the SCAN function to use the same delimiters and modifiers.

p or P adds punctuation marks to the list of characters.

q or Q ignores delimiters that are inside substrings that are enclosed in quotation marks. If the value of the *string* argument contains unmatched quotation marks, then scanning from left to right produces different words than scanning from right to left.

r or R removes leading and trailing blanks from the word that SCAN returns. If you specify the Q and R modifiers, the SCAN function first removes leading and trailing blanks from the word. Then, if the word begins with a quotation mark, SCAN also removes one layer of quotation marks from the word.

s or S adds space characters to the list of characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed).

t or T trims trailing blanks from the *string* and *charlist* arguments. If you want to remove trailing blanks from only one character argument instead of both character arguments, use the TRIM function instead of the SCAN function with the T modifier.

u or U adds uppercase letters to the list of characters.

w or W adds printable (writable) characters to the list of characters.

x or X adds hexadecimal characters to the list of characters.

Restriction This argument is supported only in SAS Viya and on the CAS server.

Tip If the *modifier* argument is a character constant, enclose the argument in quotation marks. Specify multiple modifiers in a single set of quotation marks. A *modifier* argument can also be expressed as a character variable or expression.

Details

Definitions of “Delimiter” and “Word”

A *delimiter* is any of several characters that are used to separate words. You can specify the delimiters in the *delimiter* and *modifier* arguments.

If you specify the Q modifier, delimiters inside substrings that are enclosed in quotation marks are ignored.

In the SCAN function, “word” refers to a substring that has all of these characteristics:

- It is bounded on the left by a delimiter or the beginning of the string.
- It is bounded on the right by a delimiter or the end of the string.
- It contains no delimiters.

A word can have a length of zero if there are delimiters at the beginning or end of the string, or if the string contains two or more consecutive delimiters. However, the SCAN function ignores words that have a length of zero unless you specify the M modifier.

Note: The definition of “word” is the same in the SCAN and COUNTW functions.

Using Default Delimiters in ASCII and EBCDIC Environments

If you use the SCAN function with only two arguments, then the default delimiters depend on whether your computer uses ASCII or EBCDIC characters.

- If your computer uses ASCII characters, the default delimiters are as follows:

  blank ! $ % & ( ) * + , - . / ; < ^ |

  In ASCII environments that do not contain the ^ character, the SCAN function uses the ~ character instead.
- If your computer uses EBCDIC characters, then the default delimiters are as follows:

  blank ! $ % & ( ) * + , - . / ; < ¬ | ¢

If you use the *modifier* argument without specifying any characters as delimiters, then the only delimiters that are used are delimiters that are defined by the *modifier* argument. In this case, the lists of default delimiters for ASCII and EBCDIC environments are not used. In other words, modifiers add to the list of delimiters that are explicitly specified by the *delimiter* argument. Modifiers do not add to the list of default modifiers.

The Length of the Result

Leading delimiters before the first word in the expression do not affect SCAN. If there are two or more contiguous delimiters, SCAN treats them as one.

The minimum length of the word that is returned by the SCAN function depends on whether the M modifier is specified. See "Using the SCAN Function with the M Modifier" on page 799. See also "Using the SCAN Function without the M Modifier" on page 799.

Using the SCAN Function with the M Modifier

If you specify the M modifier, the number of words in a string is defined as one plus the number of delimiters in the string. However, if you specify the Q modifier, delimiters that are inside quotation marks are ignored.

If you specify the M modifier, the SCAN function returns a word with a length of zero if one of these conditions is true:

- The string begins with a delimiter and you request the first word.
- The string ends with a delimiter and you request the last word.
- The string contains two consecutive delimiters and you request the word that is between the two delimiters.

Using the SCAN Function without the M Modifier

If you do not specify the M modifier, the number of words in a string is defined as the number of maximal substrings of consecutive non-delimiters. However, if you specify the Q modifier, delimiters that are inside quotation marks are ignored.

If you do not specify the M modifier, the SCAN function acts in these ways:

- It ignores delimiters at the beginning or end of the string.
- It treats two or more consecutive delimiters as if they were a single delimiter.

If the string contains no characters other than delimiters, or if you specify a count that is greater in absolute value than the number of words in the string, then the SCAN function returns one of the following items:

- a single blank when you call the SCAN function from a DATA step
- a string with a length of zero when you call the SCAN function from the macro processor

Using Null Arguments

The SCAN function allows character arguments to be null. Null arguments are treated as character strings with a length of zero. Numeric arguments cannot be null.

Processing SBCS and DBCS Data

The SCAN function is designed to process SBCS data, but it can also process DBCS data. Here are the criteria:

- If *expression* is not declared as VARCHAR and you are processing single-byte data, then SCAN processes SBCS.
- If *string* is declared as VARCHAR and you are processing multibyte data, then SCAN processes DBCS.

Examples

Example 1: Finding Substrings of Digits by Using the D and K Modifiers

This example finds substrings of digits. The character-list argument is null. Consequently, the list of characters is initially empty. The D modifier adds digits to the list of characters. The K modifier treats all characters that are not in the list as delimiters. Therefore, all characters except digits are delimiters.

```
proc ds2;
data test(keep=(count digits) overwrite=yes);
   dcl char(25) string digits;
   dcl double count;
   method run();
      string='Call (800) 555-1234 now!';
      do until(digits=' ');
         count+1;
         digits=scan(string, count, '', 'dko');
         output;
      end;

   end;
enddata;
run;
quit;

proc print data=test;
run;
```

Obs	digits	count
1	800	1
2	555	2
3	1234	3
4		4

Example 2: Basic SCAN Function Usage

The following statements illustrate the SCAN function:

Statements	Results
`select scan('ABC.DEF(X=Y)', 3);`	X=Y
`select scan('ABC.DEF(X=Y)', -3);`	ABC

SDF Function

Returns a survival function.

Categories: CAS
Probability

See: "CDF Function" on page 312

Syntax

SDF(*'distribution'*, *quantile*, *parameter-1, ..., parameter-k*)

Arguments

distribution
is a character string that identifies the distribution.

Note: The arguments for each of the SDF distribution functions are identical to those of the corresponding CDF distribution functions.

Here are valid distributions:

Distribution	Argument
Bernoulli	'BERNOULLI'
Beta	'BETA'
Binomial	'BINOMIAL'
Cauchy	'CAUCHY'
Chi-Square	'CHISQUARE'
Conway-Maxwell-Poisson	'CONMAXPOI'
Exponential	'EXPONENTIAL'
F	'F'
Gamma	'GAMMA'
Generalized Poisson	'GENPOISSON'

Distribution	Argument
Geometric	'GEOMETRIC'
Hypergeometric	'HYPERGEOMETRIC'
Laplace	'LAPLACE'
Logistic	'LOGISTIC'
Lognormal	'LOGNORMAL'
Negative binomial	'NEGBINOMIAL'
Normal	'NORMAL'|'GAUSS'
Normal mixture	'NORMALMIX'
Pareto	'PARETO'
Poisson	'POISSON'
T	'T'
Tweedie	'TWEEDIE'
Uniform	'UNIFORM'
Wald (inverse Gaussian)	'WALD'|'IGAUSS'
Weibull	'WEIBULL'

Note Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

quantile
is a numeric constant, variable, or expression that specifies the value of a random variable.

Data type DOUBLE

parameter-1, ..., parameter-k
are optional *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

Data type DOUBLE

Details

The SDF function computes the survival function (upper tail) from various continuous and discrete distributions. For more information, see the individual distributions listed in the table above.

The SDF function for the Conway-Maxwell-Poisson distribution has the following form:

SDF('CONMAXPOI',*y*,*λ*,*v*)

y
is a nonnegative, whole number that represents counts data.

λ
is similar to the mean, as in the Poisson distribution.

v
is a dispersion parameter.

The SDF function returns the probability that the counts value is greater than *y*.

For more information, see "Conway-Maxwell-Poisson" distribution in the PDF function on page 704.

Example

The following statements illustrate the SDF function:

Statements	Results
`select sdf('BERN', 0, .25);`	0.25
`select sdf('BETA', 0.2, 3, 4);`	0.90112
`select sdf('BINOM', 4, .5, 10);`	0.623047
`select sdf('CAUCHY', 2);`	0.147584
`select sdf('CHISQ', 11.264, 11);`	0.421419
`select sdf('CONMAXPOI', 12, 2.3, .4);`	0.197051
`select sdf('EXPO', 1);`	0.367879
`select sdf('F', 3.32, 2,3);`	0.173607
`select sdf('GAMMA', 1, 3);`	0.919699
`select sdf('GENPOISSON', .9, 1, .7);`	0.632121
`select sdf('GEOMETRIC', 5, .3);`	0.117649
`select sdf('HYPER', 2, 200, 50, 10);`	0.476327

Statements	Results
`select sdf('LAPLACE', 1);`	0.18394
`select sdf('LOGISTIC', 1);`	0.268941
`select sdf('LOGNORMAL', 1);`	0.5
`select sdf('NEGB', 1, .5, 2);`	0.5
`select sdf('NORMAL', 1.96);`	0.024998
`select sdf('NORMALMIX', 2.3, 3, .33, .33, .34, .5, 1.5, 2.5, .79, 1.6, 4.3);`	0.281869
`select sdf('PARETO', 1, 1);`	1
`select sdf('POISSON', 2, 1);`	0.080301
`select sdf('T', .9, 5);`	0.204686
`select sdf('TWEEDIE', .8, 5);`	0.408237
`select sdf('UNIFORM', 0.25);`	0.75
`select sdf('WALD', 1, 2);`	0.372302
`select sdf('WEIBULL', 1, 2);`	0.367879

See Also

Functions:

- “CDF Function” on page 312
- “LOGCDF Function” on page 622
- “LOGPDF Function” on page 626
- “LOGSDF Function” on page 628
- “PDF Function” on page 693

SEC Function

Returns the secant.

Categories: CAS
Trigonometric

Returned data type: DOUBLE

Syntax

SEC(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value that expressed in radians.

Restriction *expression* cannot be an odd multiple of PI/2.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Comparisons

The SEC function is related to the COS function:

```
sec(x)=1/cos(x)
```

Example

The following statements illustrate the SEC function:

Statement	Result
`select sec(0.5);`	`1.139494`
`select sec(0);`	`1`
`select sec(3.14159/3);`	`1.999997`

See Also

Functions:

- "COS Function" on page 388
- "COT Function" on page 390

- "CSC Function" on page 400
- "SIN Function" on page 808
- "TAN Function" on page 829

SECOND Function

Returns the second from a time or datetime value.

Categories:	Date and Time CAS
Returned data type:	DOUBLE

Syntax

SECOND(*time* | *datetime*)

Arguments

time

specifies any valid expression that represents a time value.

Data type TIME

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

datetime

specifies any valid expression that represents a datetime value.

Data type TIMESTAMP

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Example

Table: CUSTONLINE on page 1172

The following statement illustrates the SECOND function:

Statements	Results
`select second(endtime) from custonline;`	1.253 9.421 55.746 9.398 45.221 15.766 56.288 33.955 27.908 20.475

See Also

- "Dates and Times in FedSQL" on page 58

Functions:

- "DAY Function" on page 420
- "HOUR Function" on page 549
- "MINUTE Function" on page 649
- "MONTH Function" on page 655
- "YEAR Function" on page 864

SIGN Function

Returns a number that indicates the sign of a numeric value expression.

Categories:	Mathematical CAS
Returned data type:	TINYINT
Note:	This function is not supported on Teradata 14.0. and later.

Syntax

SIGN(*expression*)

Arguments

expression
specifies any valid SQL expression that evaluates to a numeric value.

Data type BIGINT, DOUBLE, FLOAT, INTEGER, REAL, SMALLINT, TINYINT

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The SIGN function returns the following values:

-1
: if *expression* < 0

0
: if *expression* = 0

1
: if *expression* > 0

Example

The following statements illustrate the SIGN function:

Statements	Results
`select sign(-5);`	`-1`
`select sign(5);`	`1`
`select sign(0);`	`0`

SIN Function

Returns the trigonometric sine.

Categories: Trigonometric
CAS

Returned data type: DOUBLE

Syntax

SIN(*expression*)

Arguments

expression

specifies any valid SQL expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Example

The following statements illustrate the SIN function:

Statements	Results
`select sin(25.6);`	`0.450441`
`select sin(5);`	`-0.95892`

See Also

Functions:

- "ARCOS Function" on page 280
- "ARSIN Function" on page 283
- "COS Function" on page 388
- "SINH Function" on page 809

SINH Function

Returns the hyperbolic sine.

Categories: Trigonometric
CAS

Returned data type: DOUBLE

Syntax

SINH(*expression*)

Arguments

expression

specifies any valid SQL expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The SINH function returns the hyperbolic sine of the argument, which is given by the following equation.

$$e^{argument} - e^{-argument}/2$$

Example

The following statements illustrate the SINH function:

Statements	Results
`select sinh(0);`	`0`
`select sinh(1);`	`1.175201`
`select sinh(-1.0);`	`-1.1752`

See Also

Functions:

- "COSH Function" on page 389
- "SIN Function" on page 808
- "TANH Function" on page 830

SKEWNESS Function

Returns the skewness of all values in an expression.

Categories:	Aggregate Descriptive Statistics CAS
Returned data type:	DOUBLE

Syntax

SKEWNESS([DISTINCT] *expression*)

Arguments

expression
specifies any valid SQL expression.

Interaction	At least three valid values are required in the column to perform the calculation. Otherwise the function returns a null value.
Data type	DOUBLE
See	"<sql-expression> Expression" on page 894 "FedSQL Expressions" on page 46

DISTINCT
(optional) eliminates duplicate rows from the aggregate calculation.

Details

Skewness is a measure of the tendency of the deviations from the mean to be larger in one direction than in the other. A positive value for skewness indicates that the data is skewed to the right. A negative value indicates that the data is skewed to the left.

Null values and SAS missing values are ignored and are not included in the computation.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each

group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Example

Table: WORLDTEMPS on page 1184

The following statement illustrates the SKEWNESS function:

Statements	Results
`select skewness(AvgHigh) from worldtemps;`	`-0.69811`

See Also

Functions:

- "STDDEV Function" on page 819

SELECT Statement Clauses:

- "SELECT Clause" on page 956
- "GROUP BY Clause" on page 968
- "HAVING Clause" on page 970

SMALLEST Function

Returns the *k*th smallest non-null or nonmissing value.

Categories:	CAS Descriptive Statistics
Returned data type:	DOUBLE

Syntax

SMALLEST(*k*, *expression* [, …*expression*])

Arguments

k

specifies any valid expression that evaluates to a numeric value to return.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

expression
specifies any valid expression that evaluates to a numeric value to be processed.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

If *k* is null or missing, less than zero, or greater than the number of values, the result is a null or missing value.

Comparisons

The SMALLEST function differs from the ORDINAL function in that SMALLEST ignores null and missing values, but ORDINAL counts null and missing values.

Example

The following statements illustrate the SMALLEST function:

Statements	Results
`select smallest(1, 456, 789, .Q, 123);`	123
`select smallest(2, 456, 789, .Q, 123);`	456
`select smallest(3, 456, 789, .Q, 123);`	789
`select smallest(4, 456, 789, .Q, 123);`	.

See Also

Functions:

- "LARGEST Function" on page 610
- "ORDINAL Function" on page 690
- "PCTL Function" on page 691

SQUANTILE Function

Returns the quantile from a distribution when you specify the right probability (SDF).

Categories:	CAS Quantile
Restriction:	This function is not supported on the CAS server.
Returned data type:	DOUBLE
See:	"SDF Function" on page 801

Syntax

SQUANTILE(*'distribution'*, *probability*, *parameter-1, ..., parameter-k*)

Arguments

distribution

is a character constant, variable, or expression that identifies the distribution.

Note: The arguments for each of the SQUANTILE Distribution functions are identical to those of the corresponding CDF Distribution functions.

Here are valid distributions:

Distribution	Argument
Bernoulli	`'BERNOULLI'`
Beta	`'BETA'`
Binomial	`'BINOMIAL'`
Cauchy	`'CAUCHY'`
Chi-Square	`'CHISQUARE'`
Conway-Maxwell-Poisson	`'CONMAXPOI'`
Exponential	`'EXPONENTIAL'`

Distribution	Argument
F	'F'
Gamma	'GAMMA'
Generalized Poisson	'GENPOISSON'
Geometric	'GEOMETRIC'
Hypergeometric	'HYPERGEOMETRIC'
Laplace	'LAPLACE'
Logistic	'LOGISTIC'
Lognormal	'LOGNORMAL'
Negative binomial	'NEGBINOMIAL'
Normal	'NORMAL'|'GAUSS'
Normal mixture	'NORMALMIX'
Pareto	'PARETO'
Poisson	'POISSON'
T	'T'
Tweedie	'TWEEDIE'
Uniform	'UNIFORM'
Wald (inverse Gaussian)	'WALD'|'IGAUSS'
Weibull	'WEIBULL'

Note: Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

probability
is a numeric constant, variable, or expression that specifies the value of a random variable.

Data type DOUBLE

parameter-1, …, parameter-k
are optional *shape*, *location*, or *scale* parameters that are appropriate for the specific distribution.

Data type DOUBLE

Details

The SQUANTILE function computes the quantile from the specified continuous or discrete distribution, based on the probability value that is provided. For more information, see the individual distributions noted in the table above.

The Conway-Maxwell-Poisson distribution of the SQUANTILE function returns the counts value *y* that is the smallest, whole number whose SDF value is less than *p*. The syntax for the Conway-Maxwell-Poisson distribution in the SQUANTILE function has the following form:

SQUANTILE('CONMAXPOI', *p*, *λ*, *v*)

p
is a real number between 0 and 1, inclusively.

λ
is similar to the mean, as in the Poisson distribution.

v
is a dispersion parameter.

For more information, see "Conway-Maxwell-Poisson" distribution in the PDF function on page 704.

Example

The following statement illustrates the SQUANTILE function:

Statements	Results
`select squantile('logistic', 1.e-20);`	`46.0517`
`select squantile('conmaxpoi',.2,2.3,.4);`	`12`

See Also

Functions:

- "CDF Function" on page 312
- "LOGCDF Function" on page 622
- "LOGPDF Function" on page 626
- "LOGSDF Function" on page 628
- "PDF Function" on page 693

- "SDF Function" on page 801
- "QUANTILE Function" on page 774

SQRT Function

Returns the square root of a value.

Categories: Mathematical
CAS

Returned data type: DOUBLE

Syntax

SQRT(*expression*)

Arguments

expression
specifies any SQL valid expression that evaluates to a nonnegative numeric value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Example

The following statements illustrate the SQRT function:

Statements	Results
`select sqrt(36);`	`6`
`select sqrt(25);`	`5`
`select sqrt(4.4);`	`2.097618`

STD Function

Returns the standard deviation.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

STD(*expression-1, expression-2* [, …*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement At least two non-null or nonmissing arguments are required. Otherwise, the function returns a null or missing value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46.

Example

The following statements illustrate the STD function:

Statements	Results
`select std(2,6);`	`2.82842712474619`
`select std(2,6,.);`	`2.82842714274619`
`select std(2,4,6,3,1);`	`1.92353840616714`

STDDEV Function

Returns the statistical standard deviation of all values in an expression.

Categories:	Aggregate Descriptive Statistics CAS
Alias:	STD
Returned data type:	DOUBLE

Syntax

STDDEV([DISTINCT] *expression*)

Arguments

expression
specifies any valid SQL expression.

Interaction	At least two valid values are required in the column to perform the calculation. Otherwise the function returns a null value.
Data type	DOUBLE
See	"<sql-expression> Expression" on page 894 "FedSQL Expressions" on page 46

DISTINCT
(optional) eliminates duplicate rows from the aggregate calculation.

Details

The standard deviation is calculated as the square root of the variance.

Null values and SAS missing values are ignored and are not included in the computation.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Example

Table: WORLDTEMPS on page 1184

The following statement illustrates the STDDEV function:

Statements	Results
`select stddev(AvgHigh) from worldtemps;`	`7.811414`

See Also

Functions:

- "STDERR Function" on page 820
- "VARIANCE Function" on page 854

SELECT Statement Clauses:

- "SELECT Clause" on page 956
- "GROUP BY Clause" on page 968
- "HAVING Clause" on page 970

STDERR Function

Returns the statistical standard error of all values in an expression.

Categories:	Aggregate Descriptive Statistics CAS
Returned data type:	DOUBLE

Syntax

STDERR([DISTINCT] *expression*)

Arguments

expression
specifies any valid SQL expression.

Interaction At least two valid values are required in the column to perform the calculation. Otherwise the function returns a null value.

Data type DOUBLE

DISTINCT
(optional) eliminates duplicate rows from the aggregate calculation.

Details

Null values and SAS missing values are ignored and are not included in the computation.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Example

Table: WORLDTEMPS on page 1184

The following statement illustrates the STDERR function:

Statements	Results
`select stderr(AvgHigh) from worldtemps;`	`2.35523`

See Also

Functions:

- “STDDEV Function” on page 819

SELECT Statement Clauses:

- “SELECT Clause” on page 956
- “GROUP BY Clause” on page 968
- “HAVING Clause” on page 970

STUDENTS_T Function

Returns the Student's t distribution of the values in an expression.

Categories:	Aggregate Descriptive Statistics CAS
Alias:	T
Returned data type:	DOUBLE

Syntax

STUDENTS_T([DISTINCT] *expression*)

Arguments

expression
specifies any valid SQL expression.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

DISTINCT
optionally eliminates duplicate rows from the aggregate calculation.

Details

The STUDENTS_T function returns the probability for the Student's t distribution with *n-1* degrees of freedom and a central distribution (nc=0).

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Comparisons

The STUDENTS_T function returns the Student's t distribution. The PROBT function returns the probability that the Student's t distribution is less than or equal to a given value.

Example

Table: DENSITIES on page 1173

The following statements illustrate the STUDENTS_T function:

Statements	Results
`select students_t(density) from densities;`	`3.565402`
`select students_t(population) from densities;`	`3.267438`

See Also

Functions:

- “PROBT Function” on page 768

SELECT Statement Clauses:

- “SELECT Clause” on page 956
- “GROUP BY Clause” on page 968
- “HAVING Clause” on page 970

SUBSTRING Function

Extracts a substring from a character string.

Categories:	Character CAS
Returned data type:	VARCHAR NVARCHAR

Syntax

SUBSTRING(*character-string* FROM *position* [FOR *length*])

Arguments

character-string
specifies any valid expression that evaluates to a character string. *character-string* is the source string that is searched for a substring.

Data type CHAR, VARCHAR, NVARCHAR

position
specifies the beginning character position.

Interaction If *position* is larger than the length of the source string, FedSQL returns a null value. For example, if you ask for a substring starting at character five but the source string is only four characters long, you get a null result.

Data type INTEGER

length
specifies the length of the substring to extract.

Interaction If *length* is zero, a negative value, or larger than the length of the expression that remains in string after position, FedSQL extracts the remainder of the expression.

Data type INTEGER

Tip If you omit length, FedSQL extracts the remainder of the expression.

Details

The SUBSTRING function returns a portion of an expression that you specify in *character-string*. The portion begins with the character that you specify by *position*, and is the number of characters that you specify in *length*.

Example

Table: "AFewWords" on page 1170

The following statement illustrates the SUBSTRING function.

Statements	Results
`select substring(word2 from 2 for 3)`	`HER` `HIN`

Statements	Results
`from afewwords;`	`ODY`

See Also

Functions:

- "TRIM Function" on page 844

SUM Function

Returns the sum of all the values in an expression.

Categories: Aggregate
Descriptive Statistics
CAS

Returned data type: The same data type as the expression

Syntax

SUM([DISTINCT] *expression*)

Arguments

expression
specifies any valid SQL expression.

Requirement The result of the SUM function must be within the range of the data type.

Data type BIGINT, DOUBLE, FLOAT, INTEGER, REAL, SMALLINT, TINYINT

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

DISTINCT
(optional) eliminates duplicate rows from the aggregate calculation.

Details

Null values and SAS missing values are ignored and are not included in the computation.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Example

Table: DENSITIES on page 1173

The following statements illustrate the SUM function:

Statements	Results
`select sum(density) from densities;`	`1728.324`
`select sum(population) from densities;`	`1.2278E8`

See Also

SELECT Statement Clauses:

- “SELECT Clause” on page 956
- “GROUP BY Clause” on page 968
- “HAVING Clause” on page 970

SUMABS Function

Returns the sum of the absolute values of the nonmissing arguments.

Categories:	CAS Descriptive Statistics
Returned data type:	DOUBLE

Syntax

SUMABS(*value-1*[, …*value-n*])

Arguments

value
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

Details

If all arguments have null or missing values, then the result is a null or missing value. Otherwise, the result is the sum of the absolute values of the nonmissing values.

Example

The following statements illustrate the SUMABS function:

Statement	Result
`select sumabs(1,.,-2,0,3,.q,-4);`	10
`select sumabs(1,3,4,3,1);`	12

SYSGET Function

Returns the value of the specified operating environment variable.

Categories: CAS
Special

Restriction: This function is supported in CAS only for users with administrative-level capabilities.

Syntax

Windows and UNIX:

SYSGET('*environment-variable*')

z/OS:

SYSGET(*operating-environment-variable*)

Arguments

environment-variable
is a character constant, variable, or expression with a value that is the name of an environment variable under Windows and UNIX.

Requirement This argument must be enclosed in single quotation marks.

operating-environment-variable
is a character constant, variable, or expression with a value that is the name of a simulated environment variable under z/OS.

Details

General Information

The SYSGET function returns the value of an environment variable as a character string. For example, this statement returns the value of the HOME environment variable under UNIX:

```
select sysget('HOME');
```

If the SYSGET function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

SYSGET Specifics under UNIX

The case of the value that you supply in the *environment-variable* argument must agree with the case of the variable that is stored in the UNIX operating environment.

SYSGET Specifics under z/OS

z/OS does not have native environment variables, but SAS supports three types of simulated environment variables that SYSGET can access.

- variables that have been created through the SET system option
- variables that have been defined in a TKMVSENV file
- under TSO, variables in the calling REXX exec or CLIST

SYSGET searches for the specified *operating-environment-variable* in each of these three locations, in the order specified in the preceding list. If the specified variable is not found in any of the locations, then the error message `NOTE: Invalid argument to the function SYSGET` is generated, and _ERROR_ is set to 1.

Names of TKMVSENV variables are case-sensitive, but names of SET, REXX, and CLIST variables are not case-sensitive.

Example: Obtain Environment Variable Values under Windows

This example obtains the value of the PATH environment variable in the Windows environment:

```
select sysget('PATH');
```

The following line is written to the SAS log.

```
C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS
\System32\WindowsPowerShell\v1.0
\;C:\Program Files\SASHome\SASFoundation\9.4\ets\sasexe;C:\Program
Files\SASHome\Secure\ccme4;C:\Program Files\SASHome\x86\Secure\ccme4;C:\Program
Files (x86)\PRISM;
```

TAN Function

Returns the tangent.

Categories:	Trigonometric CAS
Returned data type:	DOUBLE

Syntax

TAN(*expression*)

Arguments

expression

specifies any valid SQL expression that evaluates to a numeric value.

Restriction	*expression* cannot be an odd multiple of π /2
Requirement	The expression must evaluate to a value in radians.
Data type	DOUBLE
See	“<sql-expression> Expression” on page 894 “FedSQL Expressions” on page 46

Example

The following statements illustrate the TAN function:

Statements	Results
`select tan(0.5);`	`0.546302`
`select tan(3.14159/3);`	`1.732047`

See Also

Functions:

- "ATAN Function" on page 287
- "ATAN2 Function" on page 288
- "COS Function" on page 388
- "COT Function" on page 390
- "SIN Function" on page 808
- "TANH Function" on page 830

TANH Function

Returns the hyperbolic tangent.

Categories:	Trigonometric CAS
Returned data type:	DOUBLE

Syntax

TANH(*expression*)

Arguments

expression
specifies any valid SQL expression that evaluates to a numeric value.

Restriction *expression* cannot be an odd multiple of π/2

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The TANH function returns the hyperbolic tangent of the argument, which is given by the following equation.

$$\frac{e^{argument} - e^{-argument}}{e^{argument} + e^{-argument}}$$

Example

The following statements are examples of the TANH function:

Statements	Results
`select tanh(0);`	`0`
`select tanh(0.5);`	`0.462117`
`select tanh(-0.5);`	`-0.46212`

See Also

Functions:

- "ATAN Function" on page 287
- "ATAN2 Function" on page 288
- "COSH Function" on page 389
- "SINH Function" on page 809
- "TAN Function" on page 829

TIMEPART Function

Returns the time as hours, minutes, and seconds.

Categories: Date and Time

	CAS
Returned data type:	TIME

Syntax

TIMEPART(*ts*)

Arguments

ts
specifies the timestamp.

Example

The following statement illustrates the TIMEPART function:

Statements	Results
`select timepart(timestamp '2013-10-30 15:39:10');`	`15:39:10`

See Also

Functions:

- “DATEPART Function” on page 418
- “MAKEDATE Function” on page 633
- “MAKETIME Function” on page 634
- “MAKETIMESTAMP Function” on page 635

TIMEVALUE Function

Returns the equivalent of a reference amount at a base date by using variable interest rates.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

TIMEVALUE(*base-date*, *reference-date*, *reference-amount*, *compounding-interval*, *date-1*, *rate-1*[, …*date-n*, *rate-n*])

Arguments

base-date
specifies the time value of the *reference-amount* at the *base-date*.

Requirement *Base-date* is a SAS date.

Data type DOUBLE

reference-date
specifies the date of *reference-amount*.

Requirement *Reference-date* is a SAS date.

Data type DOUBLE

reference-amount
specifies the amount at the *reference-date*.

Data type DOUBLE

compounding-interval
specifies the compounding interval.

Requirement *Compounding-interval* is a SAS interval.

Data type CHAR

date
specifies the time at which *rate* takes effect. Each date is paired with a rate.

Requirement *Date* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate as numeric percentage that starts on *date*. Each rate is paired with a date.

Data type DOUBLE

Details

The following details apply to the TIMEVALUE function:

- The values for rates must be between –99 and 120.
- The list of date-rate pairs does not need to be sorted by date.

- When multiple rate changes occur on a single date, the TIMEVALUE function applies only the final rate that is listed for that date.
- Simple interest is applied for partial periods.
- There must be a valid date-rate pair whose date is at or prior to both the *reference-date* and the *base-date*.

Example

- You can express the accumulated value of an investment of $1,000 at a nominal interest rate of 10% compounded monthly for one year as the following:

```
select timevalue(date'2001-01-01', date'2000-01-01', 1000, 'month',
date'2000-01-01', 10);
```

```
1104.71306744129
```

- If the interest rate jumps to 20% halfway through the year, the resulting calculation would be as follows:

```
select timevalue(date'2001-01-01', date'2000-01-01', 1000, 'month',
date'2000-01-01', 10, date'2000-07-01', 20);
```

```
1160.63657783831
```

- The date-rate pairs do not need to be sorted by date. This flexibility allows amount_base2 and amount_base3 to assume the same value:

```
select timevalue(date'2001-01-01', date'2000-01-01', 1000, 'month',
date'2000-07-01', 20, date'2000-01-01', 10);
```

```
1160.63657783831
```

TINV Function

Returns a quantile from the *t* distribution.

Categories:	CAS Quantile
Returned data type:	DOUBLE

Syntax

TINV(*p*, *df*[, *nc*])

Arguments

p

specifies any valid expression that evaluates to a numeric probability.

Range $0 < p < 1$

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

df

specifies any valid expression that evaluates to a numeric degrees of freedom parameter.

Range $df > 0$

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

nc

specifies any valid expression that evaluates to a numeric noncentrality parameter.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The TINV function returns the pth quantile from the Student's *t* distribution with degrees of freedom *df* and a noncentrality parameter *nc*. The probability that an observation from a *t* distribution is less than or equal to the returned quantile is *p*.

TINV accepts a noninteger degree of freedom parameter *df*. If the optional parameter *nc* is not specified or is 0, the quantile from the central *t* distribution is returned.

CAUTION

For large values of nc, the algorithm can fail. In that case, a missing value is returned.

Comparisons

TINV is the inverse of the PROBT function.

Example

The following statements illustrate the TINV function:

Statement	Result
`select tinv(.95, 2);`	`2.9199855804`
`select tinv(.95, 2.5, 3);`	`11.033833625`

See Also

Functions:

- "PROBT Function" on page 768

TNONCT Function

Returns the value of the noncentrality parameter from the Student's *t* distribution.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

TNONCT(*x*, *df*, *prob*)

Arguments

x
is a numeric random variable.

Data type DOUBLE

df
is a numeric degrees of freedom parameter.

Range *df* > 0

Data type DOUBLE

prob
is a probability.

Range 0 < *prob* < 1

Data type DOUBLE

Details

The TNONCT function returns the nonnegative noncentrality parameter from a noncentral *t* distribution whose parameters are *x*, *df*, and *nc*. A Newton-type algorithm is used to find a root *nc* of the equation

$$P_t(x|df, nc) - prob = 0$$

The following relationship applies to the preceding equation:

$$P_t(x|df, nc) = \frac{1}{\Gamma\left(\frac{df}{2}\right)} \int_0^{\infty} v^{\frac{df}{2}-1} \varepsilon^{-v} \int_{-\infty}^{x\sqrt{\frac{2v}{df}}} \varepsilon^{-\frac{(u-nc)^2}{2}} du dv$$

If the algorithm fails to converge to a fixed point, a missing value is returned.

Example

The following example computes the noncentrality parameter from the *t* distribution.

```
proc ds2;
data test /overwrite=yes;
   dcl double x df nc prob;
   method init();
      x=2;
      df=4;
      do nc=1 to 3 by .5;
         prob=probt(x, df, nc);
         output;
      end;
   end;
enddata;
run;
quit;

proc print data=test;
run;

proc fedsql;
 select tnonct(x, df, prob) from test;
quit;
```

Output 4.13 *Contents of Table Test*

Obs	x	df	nc	prob
1	2	4	1.0	0.76457
2	2	4	1.5	0.61893
3	2	4	2.0	0.45567
4	2	4	2.5	0.30115
5	2	4	3.0	0.17702

Output 4.14 *Result of TNONCT Function on Table Test*

TNONCT
1
1.5
2
2.5
3

See Also

Functions:

- "CNONCT Function" on page 362
- "FNONCT Function" on page 525

TODAY Function

Returns the current date as a numeric SAS date value.

Categories:	CAS Date and Time
Returned data type:	DOUBLE

Syntax

TODAY( )

Details

The TODAY function does not take any arguments. It produces the current date in the form of a SAS date value, which is the number of days since January 1, 1960.

For more information about how FedSQL handles dates, see "Dates and Times in FedSQL" on page 58.

Example

The following statement illustrates the TODAY function:

Statements	Results
`select today();`	`19612`
`select put(td,date.);`	`11SEP13`

TRANSTRN Function

Replaces or removes all occurrences of a substring in a character string.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

TRANSTRN(*source-expression*, *target-expression*, *replacement-expression*)

Arguments

source-expression
specifies any valid expression that evaluates or can be coerced to a character string and whose characters you want to translate.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

target-expression
specifies any valid expression that evaluates or can be coerced to a character string and whose characters are searched for in *source-expression*.

Requirement The length for *target-expression* must be greater than zero.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

replacement-expression
specifies any valid expression that evaluates or can be coerced to a character string and that replaces *target-expression*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The TRANSTRN function replaces or removes all occurrences of a given substring within a character string. The TRANSTRN function does not remove trailing blanks in the *target-expression* string and the *replacement-expression* string. To remove all occurrences of *target*, specify *replacement-expression* as TRIMN(' ').

Comparisons

The TRANWRD function differs from the TRANSTRN function because TRANSTRN allows the replacement string to have a length of zero. TRANWRD uses a single blank instead when the replacement string has a length of zero.

The TRANSLATE function converts every occurrence of a user-supplied character to another character. TRANSLATE can scan for more than one character in a single call. In doing this scan, however, TRANSLATE searches for every occurrence of any of the individual characters within a string. That is, if any letter (or character) in the target string is found in the source string, it is replaced with the corresponding letter (or character) in the replacement string.

The TRANSTRN function differs from TRANSLATE in that TRANSTRN scans for substrings and replaces those substrings with a second substring.

Example

The following statements illustrate how to use the TRANSTRN function to replace occurrences of a word and to manipulate spaces.

Statements	Results
`select transtrn('Mrs. Joan Smith', 'Mrs.', 'Ms.');`	`Ms. Joan Smith`
`select transtrn('Miss Alice Cooper', 'Miss', 'Ms.');`	`Ms. Alice Cooper`
`select transtrn('CATFISH', 'FISH', 'NIP');`	`CATNIP`
`select '*' || transtrn('abcxabc', 'abc', trimn(' ')) || '*';`	`*x*`
`select '*' || transtrn('abcxabc', 'abc', ' ') || '*';`	`* x *`

See Also

Functions:

- "TRANSLATE Function" in *SAS Functions and CALL Routines: Reference*
- "TRANWRD Function" on page 841

TRANWRD Function

Replaces or removes all occurrences of a substring in a character string.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

TRANWRD(*source-expression*, *target-expression*, *replacement-expression*)

Arguments

source-expression
specifies any valid expression that evaluates or can be coerced to a character string whose characters you want to replace.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

target-expression
specifies any valid expression that evaluates or can be coerced to a character string and that is searched for in *source-expression*.

Requirement The length of the *target-expression* must be greater than zero.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

replacement-expression
specifies any valid expression that evaluates or can be coerced to a character string and that replaces *target-expression*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The TRANWRD function replaces or removes all occurrences of a given substring (or a pattern of characters) within a character string. The TRANWRD function does not remove trailing blanks in the *target-expression* string and the *replacement-expression* string.

Comparisons

The TRANWRD function differs from the TRANSTRN function because TRANSTRN allows the replacement string to have a length of zero. TRANWRD uses a single blank instead when the replacement string has a length of zero.

The TRANSLATE function converts every occurrence of a user-supplied character to another character. TRANSLATE can scan for more than one character in a single call. In doing this, however, TRANSLATE searches for every occurrence of any of the individual characters within a string. That is, if any letter (or character) in the target string is found in the source string, it is replaced with the corresponding letter (or character) in the replacement string.

The TRANWRD function differs from TRANSLATE in that it scans for substrings (or patterns of characters) and replaces those substrings with a second substring (or pattern of characters).

Example

The following statements illustrate replacing occurrences of a substring using the TRANWRD function:

Statements	Results
`select tranwrd('Whatever you so, so it with all your might. -P.T. Barnum','so','do');`	`Whatever you do, do it with all your might. -P.T. Barnum`
`select tranwrd('Believe and act as if it were impossible to pail. -Charles F. Kettering', 'pail', 'fail');`	`Believe and act as if it were impossible to fail. -Charles F. Kettering`

See Also

Functions:

- "TRANSTRN Function" on page 839

TRIGAMMA Function

Returns the value of the trigamma function.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

TRIGAMMA(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Restriction Nonpositive integers are invalid.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The TRIGAMMA function returns the derivative of the digamma function. For *expression* > 0, the TRIGAMMA function is the second derivative of the lgamma function.

Example

The following statement illustrates the TRIGAMMA function:

Statements	Results
`select trigamma(3);`	`0.394934`

See Also

Functions:

- "DIGAMMA Function" on page 431
- "LGAMMA Function" on page 616

TRIM Function

Removes leading characters, trailing characters, or both from a character string.

Categories:	Character CAS
Returned data type:	VARCHAR NVARCHAR

Syntax

TRIM([BOTH | LEADING | TRAILING] [*trim-character*] FROM *column*)

Arguments

BOTH | LEADING | TRAILING
specifies whether to remove the leading characters, the trailing characters, or both.

Default BOTH

trim-character
specifies one character to remove from *column*. Enclose a literal character in single quotation marks. If *trim-character* is not specified, the TRIM function trims all blank spaces, not just one character.

Default Blank

Data type CHAR, VARCHAR, NVARCHAR

column
is any valid expression that evaluates to a column name.

Details

The TRIM function is useful for trimming character strings of blanks or other characters before they are concatenated.

Example

Table: "AFewWords" on page 1170

The following statements illustrate the TRIM function:

Statements	Results

`select trim(word1) from afewwords;`	`*some/`
	`*every*`
	`*no*`

`select trim(both '*' from word1) from afewwords;`	`some`
	`every`
	`no`

`select trim(leading '*' from word1) from afewwords;`	`some/`
	`every*`
	`no*`

`select trim(trailing '*' from word1) from afewwords;`	`*some/`
	`*every`
	`*no`

See Also

Functions:

- “SUBSTRING Function” on page 823

TRUNC Function

Truncates a numeric value to a specified length.

Categories:	CAS Truncation
Returned data type:	DOUBLE

Syntax

TRUNC(*expression*, *length-expression*)

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Data type	DOUBLE
See	“<sql-expression> Expression” on page 894 “FedSQL Expressions” on page 46

length-expression

specifies any valid expression that evaluates to a numeric value.

Range	3–8
Data type	DOUBLE
See	“<sql-expression> Expression” on page 894 “FedSQL Expressions” on page 46

Details

The TRUNC function truncates a full-length numeric expression (stored as a DOUBLE) to a smaller number of bytes, as specified in *length-expression* and pads

the truncated bytes with 0s. The truncation and subsequent expansion duplicate the effect of storing numbers in less than full length and then reading them.

Example

The following statements illustrate the TRUNC function:

Statements	Results
`select trunc(3.1,3);`	`3.099609375`
`select trunc(3.1,4);`	`3.0999984741`
`select trunc(3.1,5);`	`3.099999994`
`select trunc(3.1,6);`	`3.1`
`select trunc(3.1,7);`	`3.1`
`select trunc(3.1,8);`	`3.1`

UPCASE Function

Converts all letters in an argument to uppercase.

Categories: CAS
Character

Alias: UPPER

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

UPCASE(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

Details

The UPCASE function copies a character expression, converts all lowercase letters to uppercase letters, and returns the altered value as a result.

Comparisons

The LOWCASE function converts all letters in an argument to lowercase letters. The UPCASE function converts all letters in an argument to uppercase letters.

Example

The following statement illustrates the UPCASE function:

Statements	Results
`select upcase('John B. Smith');`	`JOHN B. SMITH`

See Also

Functions:

- "LOWCASE Function" on page 631

URLDECODE Function

Returns a string that was decoded using the URL escape syntax.

Categories: CAS
Web Tools

Syntax

URLDECODE(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The Basics

The URL escape syntax is used to hide characters that might otherwise be significant when used in a URL.

A URL escape sequence can be one of the following:

- a plus sign, which is replaced by a blank.
- a sequence of three characters beginning with a percent sign and followed by two hexadecimal characters, which is replaced by a single character that has the specified hexadecimal value.

expression can be decoded using either SAS session encoding or UTF-8 encoding.

Operating Environment Information: In operating environments that use EBCDIC, SAS performs an extra translation step after it recognizes an escape sequence. The specified character is assumed to be an ASCII encoding. SAS uses the transport-to-local translation table to convert this character to an EBCDIC character in operating environments that use EBCDIC. For more information, see the TRANTAB option .

Character Verification

The URLDECODE and URLENCODE functions do not verify that the bytes that are produced by the escape sequences are valid characters based on the encoding.

Length of Returned Variable

If the URLDECODE function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the argument.

Example

The following statements illustrate the URLDECODE function:

Statements	Results
`select urldecode('abc+def');`	`abc def`

Statements	Results
`select urldecode('why%3F');`	`why?`
`select urldecode('%41%42%43%23%31');`	`ABC#1`

See Also

Functions:

- "URLENCODE Function" on page 850

URLENCODE Function

Returns a string that was encoded using the URL escape syntax.

Categories: CAS
Web Tools

Syntax

URLENCODE(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The Basics

expression can be encoded using either SAS session encoding or UTF-8 encoding.

The URLENCODE function encodes characters that might otherwise be significant when used in a URL. This function encodes all characters except for the following:

- all alphanumeric characters

- dollar sign ($)
- hyphen (-)
- underscore (_)
- at sign (@)
- period (.)
- exclamation point (!)
- asterisk (*)
- open parenthesis (( ) and close parenthesis ( ))
- comma (,)

Note: The encoded string might be longer than the original string. Ensure that you consider the additional length when you use this function.

Character Verification

The URLDECODE and URLENCODE functions do not verify that the bytes that are produced by the escape sequences are valid characters based on the encoding.

Example

The following statement illustrates the URLENCODE function:

Statements	Results
`select urlencode('abc def');`	`abc%20def`

See Also

Functions:

- "URLDECODE Function" on page 848

USS Function

Returns the uncorrected sum of squares of all the values in an expression.

Categories: Aggregate
Descriptive Statistics
CAS

Returned data type: DOUBLE

Syntax

USS([DISTINCT] *expression*)

Arguments

expression
specifies any valid SQL expression.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

DISTINCT
(optional) eliminates duplicate rows from the aggregate calculation.

Details

Null values and SAS missing values are ignored and are not included in the computation.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Comparisons

The CSS function returns the corrected sum of squares of all values. The USS function returns the uncorrected sum of squares.

Example

Table: DENSITIES on page 1173

The following statements illustrate the USS function:

Statements	Results
`select uss(density) from densities;`	`510190`
`select uss(population) from densities;`	`2.82E15`

See Also

Functions:

- "CSS Function" on page 401

SELECT Statement Clauses:

- "SELECT Clause" on page 956
- "GROUP BY Clause" on page 968
- "HAVING Clause" on page 970

VAR Function

Returns the variance.

Categories:	CAS Descriptive Statistics
Returned data type:	DOUBLE

Syntax

VAR(*expression-1*, *expression-2* [,…*expression-n*])

Arguments

expression

specifies any valid expression that evaluates to a numeric value. The argument list can consist of a variable list.

Requirement	At least two non-null or nonmissing arguments are required. Otherwise, the function returns a null or missing value.
Data type	DOUBLE
See	"<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Example

The following statements illustrate the VAR function:

Statements	Results
`select var(4,2,3.5,6);`	`2.7291666667`
`select var(4,6,.);`	`2`

VARIANCE Function

Returns the measure of the dispersion of all values in an expression.

Categories: Aggregate
Descriptive Statistics
CAS

Returned data type: DOUBLE

Syntax

VARIANCE([DISTINCT] *expression*)

Arguments

expression
specifies any valid SQL expression.

Interaction At least two values are required to perform the calculation. Otherwise the function returns a null value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

DISTINCT
(optional) eliminates duplicate rows from the aggregate calculation.

Details

Null values and SAS missing values are ignored and are not included in the computation.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Example

Table: DENSITIES on page 1173

The following statements illustrate the VARIANCE function:

Statements	Results
`select variance(density) from densities;`	`23498.12`
`select variance(population) from densities;`	`1.412E14`

See Also

Functions:

- “STDDEV Function” on page 819

SELECT Statement Clauses:

- “SELECT Clause” on page 956
- “GROUP BY Clause” on page 968
- “HAVING Clause” on page 970

VERIFY Function

Returns the position of the first character that is unique to an expression.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

VERIFY(*target-expression*, *search–expression*)

Arguments

target-expression

specifies any valid expression that evaluates or can be coerced to a character string that is to be searched.

Requirement Literal character strings must be enclosed in single quotation marks.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

search-expression

specifies any valid expression that evaluates or can be coerced to a character string.

Requirement Literal character strings must be enclosed in single quotation marks.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The VERIFY function returns the position of the first character in *target-expression* that is not present in *search-expression*. If there are no characters in *target-expression* that are unique from those in *search-expression*, VERIFY returns a 0.

Comparisons

The INDEX function returns the position of the first occurrence of *search-expression* that is present in *target-expression* where the VERIFY function returns the position of the first character in *target-expression* that does not contain *search-expression*.

Example

The following statement illustrates the VERIFY function:

Statements	Results
`select verify('abc', 'ab');`	`3`

See Also

Functions:

- "INDEX Function" on page 553

WEEK Function

Returns the week-number value.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

WEEK([*sas-date*], ['*descriptor*'])

Arguments

sas-date
specifies the SAS date value. If the *sas-date* argument is not specified, the WEEK function returns the week-number value of the current date.

Data type DOUBLE

descriptor
specifies the value of the descriptor. The following descriptors can be specified in uppercase or lowercase characters.

U
specifies the number-of-the-week within the year. Sunday is considered the first day of the week. The number-of-the-week value is represented as a decimal number in the range 0–53. Week 53 has no special meaning. The value of `week('31dec2013'd, 'u')` is 53. U is the default value.

Tip The U and W descriptors are similar, except that the U descriptor considers Sunday as the first day of the week, and the W descriptor considers Monday as the first day of the week.

See "The U Descriptor" on page 858

V

specifies the number-of-the-week whose value is represented as a decimal number in the range 1–53. Monday is considered the first day of the week and week 1 of the year is the week that includes both January 4 and the first Thursday of the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year.

See "The V Descriptor" on page 859

W

specifies the number-of-the-week within the year. Monday is considered the first day of the week. The number-of-the-week value is represented as a decimal number in the range 0–53. Week 53 has no special meaning. The value of `week('31dec2013'd, 'w')` is 53.

Tip The U and W descriptors are similar except that the U descriptor considers Sunday as the first day of the week, and the W descriptor considers Monday as the first day of the week.

See "The W Descriptor" on page 859

Default U

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Note If you specify a descriptor other than U, V, or W, a note is written to the SAS log to indicate that the argument is invalid. The function returns NULL.

Details

The Basics

The WEEK function reads a SAS date value and returns the week number. The WEEK function is not dependent on locale, and uses only the Gregorian calendar in its computations.

The U Descriptor

The WEEK function with the U descriptor reads a SAS date value and returns the number of the week within the year. The number-of-the-week value is represented as a decimal number in the range 0–53, with a leading zero and maximum value of 53. Week 0 means that the first day of the week occurs in the preceding year. The fifth week of the year is represented as 05.

Sunday is considered the first day of the week. For example, the value of `week('01jan2013'd, 'u')` is 0.

The V Descriptor

The WEEK function with the V descriptor reads a SAS date value and returns the week number. The number-of-the-week is represented as a decimal number in the range 01–53. The decimal number has a leading zero and a maximum value of 53. Weeks begin on a Monday, and week 1 of the year is the week that includes both January 4 and the first Thursday of the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year. In the following example, 01jan2014 and 31dec2013 occur in the same week. The first day (Monday) of that week is 30dec2013. Therefore, `week('01jan2014'd, 'v')` and `week('30dec2013'd, 'v')` both return a value of 53. This means that both dates occur in week 53 of the year 2013.

The W Descriptor

The WEEK function with the W descriptor reads a SAS date value and returns the number of the week within the year. The number-of-the-week value is represented as a decimal number in the range 0–53, with a leading zero and maximum value of 53. Week 0 means that the first day of the week occurs in the preceding year. The fifth week of the year would be represented as 05.

Monday is considered the first day of the week. Therefore, the value of `week('30dec2013'd, 'w')` is 1.

Comparisons of Descriptors

U is the default descriptor. Its range is 0-53, and the first day of the week is Sunday. The V descriptor has a range of 1-53 and the first day of the week is Monday. The W descriptor has a range of 0-53 and the first day of the week is Monday.

The following list describes the descriptors and an associated week:

- Week 0:

  U indicates the days in the current Gregorian year before week 1.

  V does not apply.

  W indicates the days in the current Gregorian year before week 1.

- Week 1:

  U begins on the first Sunday in a Gregorian year.

  V begins on the Monday between December 29 of the previous Gregorian year and January 4 of the current Gregorian year. The first ISO week can span the previous and current Gregorian years.

  W begins on the first Monday in a Gregorian year.

- End of Year Weeks:

  U specifies that the last week (52 or 53) in the year can contain less than 7 days. A Sunday to Saturday period that spans 2 consecutive Gregorian years is designated as 52 and 0 or 53 and 0.

V specifies that the last week (52 or 53) of the ISO year contains 7 days. However, the last week of the ISO year can span the current Gregorian and next Gregorian year.

W specifies that the last week (52 or 53) in the year can contain less than 7 days. A Monday to Sunday period that spans two consecutive Gregorian years is designated as 52 and 0 or 53 and 0.

Example

The following statements illustrate the values of the U, V, and W descriptors for the date August 16, 2013.

Statements	Results
`select week(date'2013-08-16', 'u');`	32
`select week(date'2013-08-16', 'v');`	33
`select week(date'2013-08-16', 'w');`	32

See Also

Functions:

- “INTNX Function” on page 582

Formats:

- “WEEKDATEw. Format” on page 181
- “WEEKDATXw. Format” on page 182
- “WEEKDAYw. Format” on page 184

WEEKDAY Function

From a SAS date value, returns a whole number that corresponds to the day of the week.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

WEEKDAY(*expression*)

Arguments

expression
specifies any valid expression that represents a SAS date value.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The WEEKDAY function produces a whole number that represents the day of the week, where 1 = Sunday, 2 = Monday, …, 7 = Saturday.

For information about how FedSQL handles date and times values, see "Dates and Times in FedSQL" on page 58.

Example

The following statement illustrates the WEEKDAY function when the current day is Sunday:

Statements	Results
`select weekday(today());`	1

WHICHC Function

Returns the first position of a character string from a list of character strings.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

WHICHC(*search-expression*, *expression-list-item-1*, *expression-list-item-2* [, …*expression-list-item-n*])

Arguments

search-expression

specifies any valid expression that evaluates or can be coerced to a character string that is compared with a list of character string expressions.

Requirement Literal character strings must be enclosed in single quotation marks.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

expression-list-item

specifies any valid expression that evaluates or can be coerced to a character string and that is a member of a list of character string expressions.

Requirements Literal character strings must be enclosed in single quotation marks.

At least two expressions are required in the list.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The WHICHC function searches the character expression list, from left to right, for the first expression that matches the search expression. If a match is found, WHICHC returns its position in the expression list. If none of the expressions match the search expression, WHICHC returns a value of 0.

Example

In the following example, 'Spain' appears twice in the list. The WHICHC function returns the first position of 'Spain' in the list:

Statements	Results
`select whichc('Spain', 'Denmark', 'Germany', 'Austria',`	`4`

Statements	Results
`'Spain', 'China', 'Egypt', 'Spain', 'France');`	

See Also

Functions:

- "WHICHN Function" on page 863

WHICHN Function

Returns the first position of a number from a list of numbers.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

WHICHN(*search-expression*, *expression-list-item-1*, *expression-list-item-2* [, …*expression-list-item-n*])

Arguments

search-expression

specifies any valid expression that evaluates to a number and that is compared with a list of numeric expressions.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

expression-list-item

specifies any valid expression that evaluates to a number and is part of a list.

Requirement At least two expressions are required in the list.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The WHICHN function searches the numeric expression list, from left to right, for the first expression that matches the search expression. If a match is found, WHICHN returns its position in the expression list. If none of the expressions match the search expression, WHICHN returns a value of 0. Arguments for the WHICHN functions can be any numeric data type.

Example

In the following example, 4.5 appears two times in the list. The WHICHN function returns the first position of 4.5 in the list.

Statements	Results
`select whichn(4.5,7.3, 8.6, 4.5, 4.5, 2.1, 6.4);`	3

See Also

Functions:

- "WHICHC Function" on page 861

YEAR Function

Returns the year from a date or datetime value.

Categories:	Date and Time CAS
Returned data type:	SMALLINT

Syntax

YEAR(*date* | *datetime*)

Arguments

date
specifies any valid expression that represents a date value.

Data type DATE

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

datetime

specifies any valid expression that represents a datetime value.

Data type TIMESTAMP

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

Example

Table: CUSTONLINE on page 1172

The following statement illustrates the YEAR function:

Statements	Results
`select year(endtime) from custonline;`	2012 2012 2012 2012 2012 2013 2013 2013 2013 2013
`select year(current_time);`	2013

See Also

- “Dates and Times in FedSQL” on page 58

Functions:

- “DAY Function” on page 420
- “HOUR Function” on page 549
- “MINUTE Function” on page 649
- “MONTH Function” on page 655
- “SECOND Function” on page 806

YIELDP Function

Returns the yield-to-maturity for a periodic cash flow stream, such as a bond.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

YIELDP(*A*, *c*, *n*, *K*, k_0, *p*)

Arguments

A

specifies the face value.

Range $A > 0$

Data type DOUBLE

c

specifies the nominal annual coupon rate, expressed as a fraction.

Range $0 \leq c < 1$

Data type DOUBLE

n

specifies the number of coupons per year.

Range $n > 0$

Data type DOUBLE

K

specifies the number of remaining coupons from settlement date to maturity.

Range $K > 0$

Data type DOUBLE

k_0

specifies the time from settlement date to the next coupon as a fraction of the annual basis.

Range $0 < k_0 \leq \frac{1}{n}$

Data type DOUBLE

p

specifies the price with accrued interest.

Range $p > 0$

Data type DOUBLE

Details

The YIELDP function is based on the following relationship:

$$P = \sum_{k=1}^{K} c(k) \frac{1}{\left(1 + \frac{y}{n}\right)^{t_k}}$$

The following relationships apply to the preceding equation:

- $t_k = nk_0 + k - 1$
- $c(k) = \frac{c}{n}A \quad for\ k = 1, \ldots, K - 1$
- $c(K) = \left(1 + \frac{c}{n}\right)A$

The YIELDP function solves for *y*.

Example

In the following example, the YIELDP function returns the yield-to-maturity of a bond that has a face value of 1000, an annual coupon rate of 0.01, 4 coupons per year, and 14 remaining coupons. The time from settlement date to next coupon date is 0.165, and the price with accrued interest is 800. The value returned is 0.077503.

```
select yieldp(1000,.01,4,14,.165,800);
```

YRDIF Function

Returns the difference in years between two dates according to specified day count conventions; returns a person's age.

Categories:	CAS Date and Time
Returned data type:	DOUBLE

Syntax

YRDIF(*start-date*, *end-date*[, *basis*])

Arguments

start-date
specifies a SAS date value that identifies the starting date.

Data type DOUBLE

end-date
specifies a SAS date value that identifies the ending date.

Data type DOUBLE

basis
identifies a character constant or variable that describes how SAS calculates a date difference or a person's age. The following character strings are valid:

'30/360'
specifies a 30-day month and a 360-day year in calculating the number of years. Each month is considered to have 30 days, and each year 360 days, regardless of the actual number of days in each month or year.

Alias '360'

Tip If either date falls at the end of a month, it is treated as if it were the last day of a 30-day month.

'ACT/ACT'
uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days that fall in 365-day years divided by 365 plus the number of days that fall in 366-day years divided by 366.

Alias 'Actual'

'ACT/360'
uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days divided by 360, regardless of the actual number of days in each year.

'ACT/365'
uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days divided by 365, regardless of the actual number of days in each year.

'AGE'
specifies that a person's age will be computed.

If you do not specify a third argument, AGE becomes the default value for *basis*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

Using YRDIF in Financial Applications

The Basics

The YRDIF function can be used in calculating interest for fixed income securities when the third argument, *basis*, is present. YRDIF returns the difference between two dates according to specified day count conventions.

Calculations That Use ACT/ACT Basis

In YRDIF calculations that use the *ACT/ACT* basis, both a 365-day year and 366-day year are taken into account. For example, if n365 equals the number of days between the start and end dates in a 365-day year, and n366 equals the number of days between the start and end dates in a 366-day year, the YRDIF calculation is computed as YRDIF=n365/365.0 + n366/366.0. This calculation corresponds to the commonly understood *ACT/ACT* day count basis that is documented in the financial literature. The values for *basis* also includes *30/360*, *ACT/360*, and *ACT/365*. Each has well-defined meanings that must be conformed to in calculating interest payments for specific financial instruments.

Computing a Person's Age

The YRDIF function can compute a person's age. The first two arguments, *start-date* and *end-date*, are required. If the value of *basis* is AGE, then YRDIF computes the age. The age computation takes into account leap years. No other values for *basis* are valid when computing a person's age.

Examples

Example 1: Calculating a Difference in Years Based on Basis

In the following example, YRDIF returns the difference in years between two dates based on each of the ACT options for *basis*.

```
select yrdif(date'1998-10-16', date'2010-02-06', '30/360') as y30360,
   yrdif(date'1998-10-16', date'2010-02-06', 'ACT/ACT') as yactact,
   yrdif(date'1998-10-16', date'2010-02-06', 'ACT/360') as yact360,
   yrdif(date'1998-10-16', date'2010-02-06', 'ACT/365') as yact365;
quit;
```

Output 4.15 *Output from ACT Calculations*

Y30360	YACTACT	YACT360	YACT365
11.30556	11.30959	11.475	11.31781

Example 2: Calculating a Person's Age

You can calculate a person's age by using the AGE option in the *basis* argument of the YRDIF function as follows:

```
select yrdif(date'1998-10-16', date'2010-02-16', 'AGE');
```

SAS returns the value 11.33699.

See Also

Functions:

- "DATDIF Function" on page 413

References

"Day count convention." *Wikipedia*, 2010. Available Day count convention. Accessed on May 1, 2013..

ISDA International Swaps and Derivatives Association, Inc "EMU and Market Conventions: Recent Developments." 1998. ISDA. Available EMU and Market Conventions: Recent Developments.

Mayle, Jan. 1994. *Standard Securities Calculation Methods – Fixed Income Securities Formulas for Analytic Measures*. Vol. 2. New York, New York: Securities Industry Association.

YYQ Function

Returns a SAS date value from year and quarter year values.

Categories:	CAS Date and Time
Returned data type:	DOUBLE

Syntax

YYQ(*year*, *quarter*)

Arguments

year

specifies any valid expression that evaluates to a two-digit or four-digit whole number that represents the year.

Interaction The YEARCUTOFF= system option defines the year value for two-digit dates.

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

quarter

specifies the quarter of the year (1, 2, 3, or 4).

Data type DOUBLE

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The YYQ function returns a SAS date value that corresponds to the first day of the specified quarter. If either *year* or *quarter* is null or missing, or if the quarter value is not valid, the result is a null or missing value.

Example

The following statements illustrate the YYQ function:

Statements	Results

`select yyq(2006,3);`
`  select put(yyq(2006,3), date7.);`
`  select put(yyq(2006,3), date9.);`
	`16983`
	`01JUL06`
	`01JUL2006`

`select yyq(2006,4);` `  select put(yyq(2006,4), date9.);`	`17075` `01OCT2006`

See Also

Concepts:

- “Dates and Times in FedSQL” on page 58

Functions:

- “QTR Function” on page 773
- “YEAR Function” on page 864

5

FedSQL Expressions and Predicates

Overview of Expressions and Predicates

Expressions are combinations of symbols and operators that FedSQL evaluates and then returns a single value. Expressions can be as simple as a single constant or column or as complex as multiple expressions joined by an operator.

This chapter focuses on three conditional expressions, CASE, COALESCE, and NULLIF, and on <sql-expression>, a generic expression that defines all other types of expressions.

Predicates specify conditions that evaluate to either true, false, or unknown. They are used most often in WHERE and HAVING clauses and in the FROM clause in join conditions.

Dictionary

BETWEEN Predicate

Selects rows where column values are within a range of values.

Syntax

expression [NOT] **BETWEEN** *expression* AND *expression*

Arguments

expression
specifies any valid SQL expression.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The BETWEEN predicate specifies a range of column values to select using these criteria:

- The SQL expressions must be of compatible data types.
- Because a BETWEEN condition evaluates the boundary values as a range, it is not necessary to specify the smaller quantity first.
- You can use the NOT logical operator to exclude a range of numbers. For example, you can use NOT to eliminate customer numbers between 1 and 15 (inclusive) so that you can retrieve data on more recently acquired customers.

Example

```
select * from invtry
   where invtry.name
      between 'A' and 'Mzzz';
```

See Also

Expressions:

- “<sql-expression> Expression” on page 894

CASE Expression

Selects result values that satisfy search conditions and value comparisons.

Syntax

CASE [*case-expression*]
WHEN *when-expression* THEN *result-expression*
…
[WHEN *when-expression* THEN *result-expression*]
[ELSE *result-expression*]
END

Arguments

case-expression
specifies any valid SQL expression that evaluates to a table column whose values are compared to *when-expression*.

See “<sql-expression> Expression” on page 894

“FedSQL Expressions” on page 46

when-expression
specifies any valid SQL search condition expression or a value expression.

- When *case-expression* is not specified, *when-expression* is a search condition expression that evaluates to true or false.
- When *case-expression* is specified, *when-expression* is an SQL value expression that is compared to *case-expression* and that evaluates to true or false.

See “<sql-expression> Expression” on page 894

result-expression
specifies an SQL expression that evaluates to a value.

See “<sql-expression> Expression” on page 894

Details

The CASE expression selects values if certain conditions are met. The *case-expression* argument returns a single value that is conditionally evaluated for each row of a table. Use the WHEN-THEN clauses to execute a CASE expression for some, but not all of the rows in the table that is being queried or created. The optional ELSE expression gives an alternative action if no THEN expression is executed.

When you omit *case-expression*, *when-expression* is evaluated as a Boolean (true or false) value. If *when-expression* returns a nonzero, non-null result, then the WHEN clause is true. If *case-expression* is specified, then it is compared with *when-expression* for equality. If *case-expression* equals *when-expression*, then the WHEN clause is true.

If the *when-expression* is true for the row that is being executed, then the *result-expression* that follows THEN is executed. If *when-expression* is false, then FedSQL evaluates the next *when-expression* until they are all evaluated. If every *when-expression* is false, then FedSQL executes the ELSE expression, and its result becomes the CASE expression's result. If no ELSE expression is present and every *when-expression* is false, then the result of the CASE expression is null.

You can use a CASE expression as an item in the SELECT clause and as either operand in an SQL expression.

Comparisons

The COALESCE expression and the NULLIF expression are variations of the CASE expression.

The following CASE expression and COALESCE expression are equivalent:

```
case
   when value1 is not null
      then value1
   when value2 is not null
      then value2
   else value3
end
```

```
coalesce(value1, value2, value3)
```

The following CASE expression and NULLIF expression are equivalent:

```
case
   when value1 = -1 then null
   else value1
end
```

```
nullif(value1, -1);
```

Examples

Example 1: The CASE Expression Using A Search Condition

Table: WORLDTEMPS on page 1184

```
select AvgLow,
   case
      when AvgLow < 32 then AvgLow + 2
      when ((AvgLow < 60) and (AvgLow > 32)) then AvgLow + 5
        when AvgLow > 60 then AvgLow + 10
        else AvgLow
   end
 as Adjusted from worldtemps;
```

SAS creates the follow table:

Output 5.1 *CASE Using a Search Condition*

avglow	ADJUSTED
45	50
33	38
17	19
56	61
57	62
28	30
51	56
75	85
36	41
33	38
25	27

Example 2: The CASE Expression Using a Value

Table: WORLDTEMPS on page 1184

```
select Country,
   case Country
```

```
      when 'Algeria' then 'Africa'
      when 'Nigeria' then 'Africa'
      when 'Netherlands' then 'Europe'
      when 'Spain' then 'Europe'
      when 'Switzerland' then 'Europe'
      when 'China' then 'Asia'
      when 'India' then 'Asia'
      when 'Venezuela' then 'South America'
       else 'Unknown'
   end
as Continent from worldtemps;
```

SAS creates the following table:

Output 5.2 *CASE Using a Value*

country	CONTINENT
Algeria	Africa
Netherlands	Europe
China	Asia
India	Asia
Venezuela	South America
Switzerland	Europe
China	Asia
Nigeria	Africa
Spain	Europe
China	Asia
Switzerland	Europe

See Also

Expressions:

- “COALESCE Expression” on page 879
- “NULLIF Expression” on page 892
- <search-condition> in the “SELECT Statement” on page 952

COALESCE Expression

Returns the first non-null value from a list of columns.

Restriction: SAS data sets, SPD Engine data sets, and SPD Server tables process null values as a blank string.

Syntax

COALESCE(*expression* [, …*expression*])

Arguments

expression
specifies any valid SQL expression.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

COALESCE accepts one or more SQL expressions of the same data type. The COALESCE expression checks the value of each SQL expression in the order in which it is listed and returns the first non-null value. If only one SQL expression is listed, the COALESCE expression returns the value of that SQL expression. If all the values of all arguments are null, the COALESCE expression returns a null value.

In some SQL DBMSs, the COALESCE expression is called the IFNULL expression.

Note: If your query contains a large number of COALESCE expressions, it might be more efficient to use a natural join instead. For more information, see "Natural Joins" on page 967.

Comparisons

The COALESCE expression is a variation of the CASE expression. For example, these two sets of code are equivalent,

```
coalesce(value1, value2, value3)
```

```
case
   when value1 is not null
```

```
      then value1
   when value2 is not null
      then value2
   else value3
end;
```

See Also

Expressions:

- "CASE Expression" on page 875

DISTINCT Predicate

Specifies that only unique rows are selected as input or output.

Syntax

Form 1: *aggregate-function* (**DISTINCT** *expression*);

Form 2: **SELECT DISTINCT** <select-list> FROM <table-expression>;

Arguments

aggregate-function
can be any aggregate function.

expression
specifies any valid SQL expression.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

SELECT <select-list> FROM <table-expression>
is a query that retrieves rows from a table.

See For more information about using the DISTINCT predicate in the SELECT statement, see the "SELECT Clause" on page 956.

Details

Both forms of the DISTINCT predicate perform a form of duplicate elimination. Use of the DISTINCT keyword with an aggregate function causes FedSQL to only

provide unique values as input to the aggregate calculation. Use of SELECT DISTINCT causes FedSQL to only return unique rows in the query's result set.

Note: Two null values are *not* considered distinct.

Example

- `select count(distinct avghigh) from worldtemps;`
-
```
select distinct c1.employee, firstname, salary
   from company as c1;
```

See Also

Statements:

- “SELECT Statement” on page 952

Functions:

- “AVG Function” on page 290
- “COUNT Function” on page 391
- “CSS Function” on page 401
- “KURTOSIS Function” on page 609
- “MAX Function” on page 642
- “MIN Function” on page 648
- “NMISS Function” on page 661
- “PROBT Function” on page 768
- “RANGE Function” on page 780
- “SKEWNESS Function” on page 811
- “STD Function” on page 818
- “STDDEV Function” on page 819
- “STDERR Function” on page 820
- “STUDENTS_T Function” on page 822
- “SUM Function” on page 825
- “USS Function” on page 851
- “VARIANCE Function” on page 854

EXISTS Predicate

Tests whether a subquery returns one or more rows.

Syntax

[NOT] **EXISTS** (*select-statement*)

Arguments

select-statement
specifies a subquery with the SELECT statement.

See "SELECT Statement" on page 952

Details

The EXISTS predicate is an operator whose right operand is a subquery. The result of an EXISTS predicate is true if the subquery resolves to at least one row. The result of a NOT EXISTS predicate is true if the subquery evaluates to zero rows.

Example

The following query subsets PAYROLL based on the criteria in the subquery. If the value for STAFF.IDNUM is on the same row as the value CT in STAFF, then the matching IDNUM in PAYROLL is included in the output. Thus, the query returns all the employees from PAYROLL who live in CT.

```
select *
   from payroll p
      where exists (select * from staff s
         where p.idnumber=s.idnum and state='CT');
```

See Also

Statements:

- "SELECT Statement" on page 952

IN Predicate

Tests set membership.

Syntax

expression [NOT] **IN** (*select-statement* | *constant* [, …*constant*])

Arguments

expression
specifies any valid SQL expression.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

select-statement
specifies a subquery with the SELECT statement.

See "SELECT Statement" on page 952

constant
specifies a number or a quoted character string (or other special notation) that indicates a fixed value. Constants are also called *literals*.

Details

The IN predicate tests whether the column value that is returned by the SQL expression on the left is a member of the set (of constants or values returned by the query expression) on the right. The IN condition is true if the value of the operand on the left is in the set of values that are defined by the operand on the right.

The NOT IN predicate negates the returned value.

Example

Table: WORLDTEMPS on page 1184

```
select city, country
   from worldtemps
      where avghigh in (90, 97);
```

SAS creates the following table:

Output 5.3 *IN Predicate Example Output Table*

city	country
Algiers	Algeria
Calcutta	India
Lagos	Nigeria

IS FALSE Predicate

Tests for a false value.

Syntax

(*expression*) **IS** [NOT] **FALSE**

Arguments

expression
specifies any valid SQL expression.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

IS FALSE is a predicate that tests for a false value. IS FALSE is used in the WHERE, ON, and HAVING clauses. The IS FALSE predicate resolves to true if the result of the SQL expression is false and resolves to false if it is true.

Comparisons

The IS TRUE predicate tests for true values.

Example

Table: WORLDCITYCOORDS on page 1182

```
select city
   from worldcitycoords
      where (latitude = 40) is false;
```

SAS creates the following table:

Output 5.4 *IS FALSE Example Output Table*

city
Algiers
Shanghai
Hong Kong
Bombay
Calcutta
Amsterdam
Lagos
Zurich
Caracas

See Also

Predicates:

- "IS TRUE Predicate" on page 888
- "IS UNKNOWN Predicate" on page 890
- <search-condition> in the "SELECT Statement" on page 952

IS MISSING Predicate

Tests for a SAS missing value in a SAS native data store.

Syntax

expression **IS** [NOT] **MISSING**

Arguments

expression
specifies any valid SQL expression.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

IS MISSING is a predicate that tests for a SAS missing value. IS MISSING is used in the WHERE, ON, and HAVING clauses. The IS MISSING predicate resolves to true if the result of the SQL expression is a SAS missing value and resolves to false if it is not a SAS missing value.

The IS MISSING predicate is valid only in use with SAS native data stores. Only DOUBLE and CHAR data types support missing values.

Comparisons

The IS NULL predicate tests for null values.

Example

Table: WORLDCITYCOORDS on page 1182

```
select *
    from worldcitycoords
      where city is missing;
```

SAS creates the following table:

Output 5.5 *IS MISSING Example Output Table*

city	country	latitude	longitude
	China	40	116

See Also

Predicates:

- "IS NULL Predicate" on page 887
- <search-condition> in the "SELECT Statement" on page 952

IS NULL Predicate

Tests for a null value.

Syntax

expression **IS** [NOT] **NULL**

Arguments

expression
specifies any valid SQL expression.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

IS NULL is a predicate that tests for a null value. IS NULL is used in the WHERE, ON, and HAVING clauses. The IS NULL predicate resolves to true if the result of the SQL expression is null and resolves to false if it is not null.

Comparisons

The IS MISSING predicate tests for SAS missing values in SAS native data stores.

Example

Table: WORLDCITYCOORDS on page 1182

```
select city
   from worldcitycoords
```

```
where latitude is not null;
```

SAS creates the following table:

Output 5.6 *IS NULL Example Output Table*

city
Algiers
Shanghai
Hong Kong
Bombay
Calcutta
Amsterdam
Lagos
Madrid
Zurich
Caracas

See Also

Predicates:

- "IS MISSING Predicate" on page 885
- <search-condition> in the "SELECT Statement" on page 952

IS TRUE Predicate

Tests for a true value.

Syntax

(*expression*) **IS** [NOT] **TRUE**

Arguments

expression
specifies any valid SQL expression.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

IS TRUE is a predicate that tests for a true value. IS TRUE is used in the WHERE, ON, and HAVING clauses. The IS TRUE predicate resolves to true if the result of the SQL expression is true and resolves to false if it is false.

Comparisons

The IS FALSE predicate tests for false values.

Example

Table: WORLDCITYCOORDS on page 1182

```
select city
   from worldcitycoords
      where (latitude = 40) is true;
```

SAS creates the following table:

Output 5.7 *IS TRUE Example Output*

city
Madrid

See Also

Predicates:

- "IS FALSE Predicate" on page 884
- "IS UNKNOWN Predicate" on page 890

IS UNKNOWN Predicate

Tests for an unknown value.

Syntax

expression **IS** [NOT] **UNKNOWN**

Arguments

expression
specifies any valid SQL expression.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

IS UNKNOWN is a predicate that tests for an unknown value. IS UNKNOWN is used in the WHERE, ON, and HAVING clauses. The IS UNKNOWN predicate resolves to true if the result of the SQL expression is unknown and resolves to false if it is a valid value.

See Also

Predicates:

- "IS FALSE Predicate" on page 884
- "IS TRUE Predicate" on page 888
- <search-condition> in the "SELECT Statement" on page 952

LIKE Predicate

Tests for a matching pattern.

Syntax

expression [NOT] **LIKE** *expression*

Arguments

expression
specifies any valid SQL expression that is either a character string type or a binary string type.

Tip The SQL expression on the right side of the syntax, that is the pattern, is most likely to be a literal.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

Overview of the LIKE Predicate

The LIKE predicate selects rows by comparing character strings with a pattern-matching specification. It resolves to true and displays the matched string or strings if the left operand matches the pattern that is specified by the right operand.

Escape characters are not supported.

Note: If no rows are returned, the result is a null value.

Patterns for Searching

Patterns include three classes of characters:

underscore (_)
matches any single character.

percent sign (%)
matches any sequence of zero or more characters.

any other character
matches that character.

These patterns can appear before, after, or on both sides of characters that you want to match. The LIKE condition is case-sensitive.

The following list uses these values: `Smith`, `Smooth`, `Smothers`, `Smart`, and `Smuggle`.

`'Sm%'`
matches `Smith`, `Smooth`, `Smothers`, `Smart`, `Smuggle`.

`'%th'`
matches `Smith`, `Smooth`.

`'S__gg%'`
matches `Smuggle`.

`'S_o'`
matches a three-letter word, so it has no matches here.

`'S_o%'`
matches `Smooth`, `Smothers`.

`'S%th'`
matches `Smith`, `Smooth`.

`'M'`
matches the single, uppercase character `m` only, so it has no matches here.

Searching for Mixed-Case Strings

To search for mixed-case strings, use the UPPER function to make all the names uppercase before entering the LIKE condition:

```
upper(name) like 'SM%';
```

Note: When you are using the % character, be aware of the effect of trailing blanks. You might have to use the TRIM function to remove trailing blanks in order to match values.

Example

Table: DENSITIES on page 1173

```
select name, population
   from densities
      where name like 'Al%';
```

See Also

Functions:

- "TRIM Function" on page 844

NULLIF Expression

Returns a null value if the two specified expressions are equal; otherwise, returns the first expression.

Restriction: The BASE file format processes a null value as DOUBLE values in some situations and as a blank string in other situations. For more information, see "How FedSQL Processes Nulls and SAS Missing Values" on page 20.

Syntax

NULLIF(*expression-1, expression-2*)

Arguments

expression
specifies any valid SQL expression.

Data type All data types are valid.

See "<sql-expression> Expression" on page 894

"FedSQL Expressions" on page 46

Details

The NULLIF expression compares two SQL expressions and, if they are equal, returns a null value. The NULLIF expression enables you to replace a missing or inapplicable value with a null value and to use SQL's behavior for null values.

Comparisons

The NULLIF expression is a shorthand syntax for a special CASE expression. For example, if a student misses a test, a -1 is entered in the GRADES table. To replace this -1 with a null value, you could use the following CASE code.

```
update grades
   set testscore =
    CASE
      when testscore = '-1' then null
      ELSE testscore
     END;
```

The following code uses the shorter NULLIF expression.

```
update grades
   set testscore = NULLIF(testscore, '-1');
```

The IFNULL function compares two SQL expressions and returns the second SQL expression if the first SQL expression is a null value. The NULLIF expression compares two SQL expressions and returns a null value if the two SQL expressions are equal.

Example

Table: WORLDCITYCOORDS on page 1182

```
missingLong= '.L';
```

```
update worldcitycoords
set longitude = nullif(missingLong, '.');
select city
   from worldcitycoords
      where Longitude='.L';
```

See Also

Expressions:

- "CASE Expression" on page 875
- "COALESCE Expression" on page 879

Functions:

- "IFNULL Function" on page 551

<sql-expression> Expression

Produces a single value from a combination of symbols and operators or predicates.

Syntax

<sql-expression>::=
constant
| [*alias*] *column*
| *function*
| (*scalar-subquery*)
| (<sql-expression>)
| <sql-expression> {*operator* | *predicate*} <sql-expression>

Arguments

constant
is a number, a quoted character string, or a datetime value that represents a single, specific data value.

alias
is the alias that is assigned to a table by using the AS keyword in the FROM clause of a SELECT statement.

column
is the name of a column.

function
is a SAS or aggregate function.

See "FedSQL Functions" on page 207

scalar-subquery
is a subquery that returns a single value.

operator
is a symbol that specifies an action that is performed on one or more expressions. The following table shows valid operators. An expression can also contain the CASE or COALESCE expressions. For more information, see "CASE Expression" on page 875 or "COALESCE Expression" on page 879.

Table 5.1 *Valid Operators*

Operator	Description
+	adds
–	subtracts
*	multiplies
/	divides
=	equals
<>	does not equal
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to
**	raises to a power
unary –	indicates a negative number
||	concatenates

predicate
is an expression that returns true, false, or unknown.

Valid predicates are as follows.

- "BETWEEN Predicate" on page 874
- "DISTINCT Predicate" on page 880
- "EXISTS Predicate" on page 882
- "IN Predicate" on page 883
- "IS FALSE Predicate" on page 884

Details

Overview of <sql-expression>

Simple expressions can be a single constant, column name, or function. Complex expressions are two or more simple expressions that are joined by an operator or predicate.

Functions in Expressions

An expression can contain a SAS function or an aggregate function. SAS functions perform a computation or system manipulation on one or more arguments and return a value. Aggregate functions produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. If GROUP BY is omitted, then all the rows in the table are considered to be a single group. Aggregate functions reduce all the values in each row or column in a table to one summarizing or aggregate value. For example, the sum (one value) of a column results from the addition of all the values in the column.

Subqueries in Expressions

FedSQL allows a scalar subquery (contained in parentheses) at any point in an expression where a simple column value or constant can be used. In this case, a subquery must return a single value (that is, one row with only one column).

Order of Evaluation

The operators and predicates that are shown in the following table are listed in the order in which they are evaluated.

Table 5.2 *Expressions, Operators, and Predicates and Order of Evaluation*

Group	Expressions, Operators, and Predicates	Description
0	( )	forces the expression enclosed to be evaluated first
1	CASE expression	See "CASE Expression" on page 875

Group	Expressions, Operators, and Predicates	Description
2	**	raises to a power
	unary +, unary −	indicates a positive or negative number
3	*	multiplies
	/	divides
4	+	adds
	−	subtracts
5	||	concatenates
6	[NOT] BETWEEN predicate	See “BETWEEN Predicate” on page 874
	DISTINCT predicate	See “DISTINCT Predicate” on page 880
	[NOT] EXISTS predicate	See “EXISTS Predicate” on page 882
	[NOT] IN predicate	See “IN Predicate” on page 883
	IS [NOT] TRUE predicate	See “IS TRUE Predicate” on page 888
	IS [NOT] FALSE predicate	See “IS FALSE Predicate” on page 884
	IS [NOT] MISSING predicate	See “IS MISSING Predicate” on page 885
	IS [NOT] NULL predicate	See “IS NULL Predicate” on page 887
	IS [NOT] UNKNOWN predicate	See “IS UNKNOWN Predicate” on page 890
	LIKE predicate	See “LIKE Predicate” on page 890
7	=	equals
	^=, <>	does not equal
	>	is greater than
	<	is less than

Group	Expressions, Operators, and Predicates	Description
	>=	is greater than or equal to
	<=	is less than or equal to
8	AND	indicates logical AND
9	OR	indicates logical OR
10	NOT	indicates logical NOT

SAS missing values and null values always appear as the smallest value in the collating sequence.

You can use parentheses to group values or to nest mathematical expressions. Parentheses make expressions easier to read and can also be used to change the order of evaluation of the operators. Evaluating expressions with parentheses begins at the deepest level of parentheses and moves outward. For example, SAS evaluates A+B*C as A+(B*C), although you can add parentheses to make it evaluate as (A+B)*C for a different result.

See Also

Statements:

- “SELECT Statement” on page 952
- “FedSQL Expressions” on page 46

6

FedSQL Informats

Definition of an Informat

An *informat* is an instruction that determines how values are read into a column. For example, the following value contains a dollar sign and commas:

```
$1,000,000
```

To remove the dollar sign ($) and commas (,) before storing the numeric value 1000000 in a variable, read this value with the COMMA11. informat.

General Informat Syntax

FedSQL informats have the following syntax:

[$]*informat*[*w*].[*d*]

Arguments

$
indicates a character informat; its absence indicates a numeric informat.

informat
names the informat. The informat is a SAS informat or a user-defined informat that was previously defined with the INVALUE statement in PROC FORMAT. For more information on user-defined informats, see PROC FORMAT in *Base SAS Procedures Guide*.

w
specifies the informat width, which for most informats is the number of columns in the input data.

d
specifies an optional decimal scaling factor in the numeric informats. SAS divides the input data by 10 to the power of *d*.

Note: Even though SAS can read up to 31 decimal places when you specify some numeric informats, floating-point numbers with more than 12 decimal places might lose precision because of the limitations of the eight-byte, floating-point representation used by most computers.

Informats always contain a period (.) as a part of the name. If you omit the *w* and the *d* values from the informat, SAS uses default values. If the data contains decimal points, SAS ignores the *d* value and reads the number of decimal places that are actually in the input data.

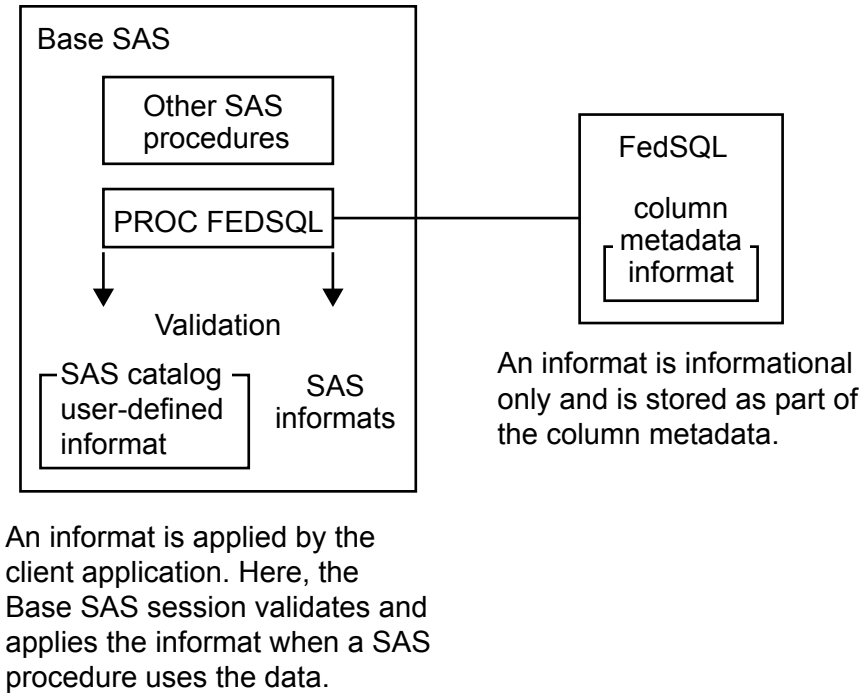
For more information about how informats work and a complete list of informats, see *SAS Formats and Informats: Reference*.

How Informats Are Used in FedSQL

Informats are informational and do not operate on your data at run time in the FedSQL environment. The client application is responsible for using the informat to convert the raw data when the table is returned to the application.

Note: If the informat that is stored in the FedSQL environment is invalid, an error occurs only when the invalid informat is used in the client application.

Figure 6.1 Diagram of How Informats Work



However, in FedSQL, you can store and retrieve informat names. The informat name is associated with a column—either temporarily for the duration of an operation or permanently-by storing the informat as a metadata attribute on the column. The metadata then can be retrieved for subsequent operations.

FedSQL supports SAS informats as follows.

- Both informats supplied by SAS and user-defined informats can be associated with a column. For information about how to create your own informat in SAS, see PROC FORMAT in *Base SAS Procedures Guide*.

  Note: To create and access user-defined informats, a Base SAS session must be available in order to access the SAS catalog file that stores the SAS informat definitions.

- Only the SAS data sets, SPD Engine data sets, and SPD Server tables support storing and retrieving an informat with a column.
- Informats can be associated with all data types, but all data types are converted to either CHAR or DOUBLE.
- You can associate SAS informats with a column by using the HAVING clause of the FedSQL CREATE TABLE or the SET clause of the FedSQL ALTER TABLE statement. For more information, see "How to Specify Informats in FedSQL" on page 902.

For more information and a complete list of informats that are supplied by SAS, see the section on informats in *SAS Formats and Informats: Reference*.

How to Specify Informats in FedSQL

In FedSQL, specify informats as an attribute in the HAVING clause of the CREATE TABLE statement. For example, in the following statement, the column X is declared with the IEEE8.2 format and the BITS5.2 informat.

```
create table a (x double having format ieee8.2 informat bits5.2 label 'foo');
```

Validation of FedSQL Informats

Informats are not validated by a data source or applied to a column until execution time.

When a table is created using FedSQL, no validation occurs.

When metadata for a column is requested, the informat name is returned without validation.

FedSQL Informat Example

```
create table y (x double having label 'price' format $5. informat
$charzb4.3);
```

7

FedSQL Statements

Overview of Statements

A FedSQL statement is a series of items that can include keywords, identifiers, special characters, and operators. All FedSQL statements end with a semicolon.

There are five categories of statements:

data definition
: statements that are used to create, modify, and delete tables and views.

data control
: statements that are used to control transactions with the database.

data manipulation
: statements that are used to add, update, and delete data.

data retrieval
: statements that are used to retrieve data, or retrieve information about a data table or a SQL view.

SQL pass-through facility
: statements that connect to a DBMS and enable you to send DBMS-specific SQL statements directly to the DBMS.

Some statements are included in more than one category.

FedSQL Statements by Category

Category	Language Elements	Description
Data Control	BEGIN Statement (p. 911)	Marks the beginning of a transaction that comprises multiple statements.
	COMMIT Statement (p. 912)	Makes changes that have been performed since the start of a transaction a permanent part of the database.
	ROLLBACK Statement (p. 951)	Rolls back transaction changes to the beginning of the transaction.
Data Definition	ALTER TABLE Statement (p. 905)	Adds or drops table columns and modifies column definitions.
	CREATE INDEX Statement (p. 913)	Creates an index on columns in a specified table.
	CREATE TABLE Statement (p. 915)	Creates a new table.
	CREATE VIEW Statement (p. 934)	Creates a FedSQL view of data from one or more tables or other views.
	DESCRIBE TABLE Statement (p. 937)	Retrieves SQL from a table and returns a result set.
	DESCRIBE VIEW Statement (p. 940)	Retrieves SQL from a view and returns a result set.
	DROP INDEX Statement (p. 941)	Removes the specified index from a table.
	DROP TABLE Statement (p. 942)	Removes a table from the database.
	DROP VIEW Statement (p. 944)	Removes a view from the database.

Category	Language Elements	Description
	SELECT Statement (p. 952)	Retrieves columns and rows of data from tables.
Data Manipulation	DELETE Statement (p. 936)	Deletes rows from a table.
	INSERT Statement (p. 948)	Adds rows to a specified table.
	SELECT Statement (p. 952)	Retrieves columns and rows of data from tables.
	UPDATE Statement (p. 984)	Modifies a column's values in existing rows of a table.
Data Retrieval	DESCRIBE TABLE Statement (p. 937)	Retrieves SQL from a table and returns a result set.
	DESCRIBE VIEW Statement (p. 940)	Retrieves SQL from a view and returns a result set.
	SELECT Statement (p. 952)	Retrieves columns and rows of data from tables.
FedSQL pass-through	EXECUTE Statement (p. 945)	Sends a DBMS-specific statement to a DBMS that FedSQL supports.

Dictionary

ALTER TABLE Statement

Adds or drops table columns and modifies column definitions.

Category: Data Definition

Restrictions: This statement is not supported in FedSQL programs that run in CAS.

Constraint syntax is defined by the data source. The constraint syntax that is provided here is a general syntax. For complete constraint syntax, see the documentation for your data source.

Data source: SAS data set, Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, Hive, Impala, JDBC, MDS, Microsoft SQL Server, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Spark, Teradata, Vertica, Yellowbrick

Notes: Braces in the syntax convention indicate a syntax grouping. The escape character (\ ) before a brace indicates that the brace is required in the syntax. Table options must be contained by braces ({ }).

SAS 9.4, SAS Viya, and SAS Federation Server can support different data sources. For support information: SAS 9.4 and SAS Viya users, see "Data Source Support" in *Base SAS Procedures Guide*; SAS Federation Server users, see SAS Federation Server Documentation.

Syntax

ALTER TABLE *table* [\{OPTIONS *SAS-table-option=value* [, …*SAS-table-option=value*] \}]

ADD COLUMN <column-definition> [, …<column-definition>]

| ADD CONSTRAINT <table-constraint>

| ALTER [COLUMN] <column-definition> [, …<column-definition>]

| DROP COLUMN *column* [, …*column*] [FORCE]

| DROP CONSTRAINT *constraint* [, …*constraint*]

| RENAME TO *table*

| RENAME *column* TO *column*

;

<column-definition>::=

column data-type [<column-constraint>] [SET DEFAULT *value* | DROP DEFAULT]

<column-constraint>::=

CONSTRAINT *constraint*

{ CHECK (*search-condition*)

| PRIMARY KEY

| UNIQUE

| NOT NULL }

<table-constraint>::=

CONSTRAINT *constraint*

{ CHECK (*search-condition*)

| PRIMARY KEY (*column* [, …*column*])

| UNIQUE (*column* [, …*column*])

| <referential-constraint> }

[<constraint-check-time>]

<referential-constraint>::=

FOREIGN KEY (*referencing-column* [, … *referencing-column*])

REFERENCES *referenced-table* (*referenced-column* [, …*referenced-column*])

[<referential-trigger-action>]

<referential-trigger-action>::=

{ [ON UPDATE <referential-action> [ON DELETE <referential-action>]] }

| { [ON DELETE <referential-action> [ON UPDATE <referential-action>]] }

<referential-action>::=

CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION

<constraint-check-time>::=

{ DEFERRABLE [INITIALLY DEFERRED | INITIALLY IMMEDIATE] }

| { [INITIALLY DEFERRED | INITIALLY IMMEDIATE] DEFERRABLE }

| INITIALLY DEFERRED

| { NOT DEFERRABLE [INITIALLY IMMEDIATE] }

| { [INITIALLY IMMEDIATE] NOT DEFERRABLE }

Arguments

table

specifies a table to modify. *table* can be specified in one of these forms:

- *catalog.schema.table-name*
- *schema.table-name*
- *catalog.table-name*
- *table-name*

catalog

is an implementation of the ANSI SQL standard for an SQL catalog. The catalog is a data container object that groups logically related schemas. It is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.

schema

is an implementation of the ANSI SQL standard for an SQL schema. The schema is a data container object that groups files such as tables and views and other objects that are supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

table-name

is the name of the table.

Restriction You cannot alter an MDS table while it is referenced in another transaction or statement. If a request fails, make sure other users are no longer using the table or have disconnected.

Requirement Table naming conventions are based on the data source. When more than one data source is involved, the maximum length of a table name is determined by the maximum length that is supported by all of the data sources and FedSQL. For example, if your data sources are a SAS data set that has a maximum of 32 characters and MySQL that has a maximum of 64 characters, the maximum length of a table name is 32 characters. For more information, see the documentation for your data source.

{OPTIONS *SAS-table-option=value* [, …*SAS-table-option=value*]}

specifies one or more table options and their respective values to apply to the table.

Requirement The OPTION argument and all table options must be enclosed in braces ({ }).

See “FedSQL Statement Table Options” on page 987

column

specifies the name of a column in a table.

Requirement Each column in a table must be unique.

ADD COLUMN <column-definition> [, …<column-definition>]

specifies to add a column to a table.

Data source Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, Hive, Impala, JDBC, MDS, Microsoft SQL Server, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Spark, Teradata, Vertica, Yellowbrick

See column on page 907 and "<column-definition> Arguments" on page 918 in the CREATE TABLE statement

Example "Adding and Deleting Columns" on page 909

ADD CONSTRAINT <table-constraint>
specifies to add an integrity constraint to one or more columns.

Data source SAS data set, Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, Microsoft SQL Server, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Teradata, Vertica, Yellowbrick

Note Constraint validation and enforcement is performed by the data source. You should be familiar with data source's requirements before adding or dropping an integrity constraint.

See "<column-constraint> and <table-constraint> Arguments" on page 927 in the CREATE TABLE statement

Example "Adding and Deleting Constraints" on page 909

ALTER [COLUMN] <column-definition> [, …<column-definition>]
specifies to modify the definition of one or more columns.

Data source Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, MDS, Microsoft SQL Server, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP IQ, Teradata, Vertica, Yellowbrick

See "<column-definition> Arguments" on page 918 in the CREATE TABLE statement

Example "Altering Column Definitions" on page 910

DROP COLUMN *column* [, …*column*] [FORCE]
specifies to delete the specified column from the table. When FORCE is specified, the column is dropped from the table without error processing. Use the FORCE keyword only when you are certain that dropping the column without error processing will not negatively affect the table.

Data source Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, Hive, JDBC, MDS, Microsoft SQL Server, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Teradata, Vertica, Yellowbrick

See "*column*" on page 907

DROP CONSTRAINT *constraint* [, …*constraint*]
specifies to delete an integrity constraint.

constraint
specifies the name of the constraint to drop.

Data source	SAS data set, Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, Microsoft SQL Server, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Teradata, Vertica, Yellowbrick

RENAME
changes the specified table or column name to a new name. The new name follows the TO keyword.

Data source	Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, MDS, Microsoft SQL Server, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Teradata, Vertica, Yellowbrick

<column-definition> Arguments

See “<column-definition> Arguments” on page 918 in the CREATE TABLE statement.

<column-constraint> and <table-constraint> Arguments

See “<column-constraint> and <table-constraint> Arguments” on page 927 in the CREATE TABLE statement.

Details

Adding and Deleting Columns

To add a column to a table, use the ALTER TABLE statement with the ADD COLUMN clause. By using the ADD COLUMN clause, you name the column and define the column attributes, such as the column data type, a default value, and a label for the column.

```
alter table personal_info add column em_num char(16);
```

To delete a column, you need to specify only the column name.

```
alter table personal_info drop em_num;
```

Adding and Deleting Constraints

You can define a constraint on a column that is added with the ADD COLUMN clause in all data sources that support integrity constraints as follows:

```
alter table mytest add y char(5) constraint c2 unique;
```

Exceptions are SAP HANA and Vertica. For these data sources, you must add the column first, and then add the constraint with ADD CONSTRAINT. In addition, for Vertica you must specify a constraint name when adding a UNIQUE constraint with the ALTER TABLE statement.

By using the ADD CONSTRAINT clause, you can add a constraint for a table, or you can add or modify a constraint for one or more existing columns. The following code adds a primary key constraint to a table with ALTER TABLE.

```
alter table customers add constraint pkey primary key(custid);
```

The following code adds a CHECK constraint to an existing column.

```
alter table mytest add constraint chk check (col1 < 5);
```

You delete a constraint by using the DROP CONSTRAINT clause. The following code is an example of deleting a constraint from a SAS data set. The code deletes a primary key constraint that was defined at table creation. When they are defined at table creation, primary keys in a SAS data set are created with the default name _pk0001_.

```
alter table customers drop constraint _pk0001_;
```

When you add or modify constraints, you must know the constraint syntax that the data source supports. Constraint validation is performed by the data source.

Altering Column Definitions

To alter a column definition, use the ALTER TABLE statement with the ALTER clause.

```
alter table personal_info alter country set default 'UK';
```

This example drops a default value from a column.

```
alter table personal_info alter country drop default;
```

This example renames a column.

```
alter table sales rename column y to customers;
```

Comparisons

You use the ALTER TABLE statement to alter a table after it has been created. Use the CREATE TABLE statement to create a table.

See Also

- “How to Store, Change, Delete, and Use Stored Formats ” on page 78

Statements:

- “CREATE TABLE Statement” on page 915
- “DROP INDEX Statement” on page 941

BEGIN Statement

Marks the beginning of a transaction that comprises multiple statements.

Category: Data Control

Restrictions: This statement is not supported in FedSQL programs that run in CAS.

The BEGIN statement has an effect only when autocommit functionality is off.

Data source: Greenplum, HAWQ, MDS, Microsoft SQL Server, MySQL, ODBC, Oracle, SAP HANA, SAP IQ, Teradata, Vertica

Note: SAS 9.4, SAS Viya, and SAS Federation Server can support different data sources. For support information: SAS 9.4 and SAS Viya users, see "Data Source Support" in *Base SAS Procedures Guide*; SAS Federation Server users, see SAS Federation Server Documentation.

Syntax

BEGIN [TRANSACTION];

Details

The BEGIN statement marks a point at which the data is logically and physically consistent. If errors are encountered during a transaction, the user can roll back all changes to the data to this last known state of consistency.

When you use FedSQL processing, this statement is not really necessary. By default, autocommit functionality is on, which means that all updates are committed immediately after each request is submitted, and no rollback is possible. A transaction is effectively started with each update. The BEGIN statement is provided to enable you to mark the beginning of a transaction that comprises multiple statements.

See the server administration documentation for information about how to turn off autocommit functionality. For example, see *SAS Federation Server: Administrator's Guide* for the appropriate connection option to the FedSQL driver. For the FEDSQL procedure, see *Base SAS Procedures Guide*.

See Also

Statements:

- "COMMIT Statement" on page 912
- "ROLLBACK Statement" on page 951

COMMIT Statement

Makes changes that have been performed since the start of a transaction a permanent part of the database.

Category:	Data Control
Restrictions:	This statement is not supported in FedSQL programs that run in CAS. The COMMIT statement has an effect only when autocommit functionality is off.
Data source:	Greenplum, HAWQ, MDS, Microsoft SQL Server, MySQL, ODBC, Oracle, SAP HANA, SAP IQ, Teradata, Vertica
Note:	SAS 9.4, SAS Viya, and SAS Federation Server can support different data sources. For support information: SAS 9.4 and SAS Viya users, see "Data Source Support" in *Base SAS Procedures Guide*; SAS Federation Server users, see SAS Federation Server Documentation.

Syntax

COMMIT [TRANSACTION];

Details

When your program has completed all of the statements in the transaction, you must explicitly terminate the transaction using COMMIT or ROLLBACK. You use a COMMIT statement to make the changes to the database permanent.

You cannot roll back the changes to the database after the COMMIT statement is executed.

Note: The COMMIT statement has an effect only when autocommit functionality is off. In most data sources, autocommit functionality is on by default. Refer to the server administration documentation for information about how to turn off autocommit functionality. For example, see the *SAS Federation Server: Administrator's Guide* for the appropriate connection option to the FedSQL driver. For the FEDSQL procedure, see the *Base SAS Procedures Guide*.

Comparisons

The ROLLBACK statement causes all the uncommitted changes that were made by the transaction to be rolled back to the start of the transaction or to the point that is marked by the BEGIN statement. The COMMIT statement takes all the data changes that have been performed since the start of the transaction and makes them a permanent part of the database.

See Also

Statements:

- "BEGIN Statement" on page 911
- "ROLLBACK Statement" on page 951

CREATE INDEX Statement

Creates an index on columns in a specified table.

Category:	Data Definition
Restriction:	This statement is not supported in FedSQL programs that run in CAS.
Data source:	SAS data set, SPD Engine data set, SPD Server table, Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, JDBC, MDS, Microsoft SQL Server, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Teradata
Note:	SAS 9.4, SAS Viya, and SAS Federation Server can support different data sources. For support information: SAS 9.4 and SAS Viya users, see "Data Source Support" in *Base SAS Procedures Guide*; SAS Federation Server users, see SAS Federation Server Documentation.

Syntax

CREATE [UNIQUE] **INDEX** *index* ON *table*

(*column* [ASCENDING | DESCENDING][, …*column*]);

Arguments

UNIQUE
creates a unique index on a table.

index
specifies the name of the index.

Restriction *For SAS data sets* – If you are creating an index on one column only, then *index* must be the same as *column*. If you are creating an index on more than one column, then *index* cannot be the same as any column in the table.

table
specifies the name of the table that contains the column or columns to be indexed.

column
specifies the name of the column to which the index applies. Specify two or more column names to create a composite index.

Tip If you search two or more columns as a unit or if you have queries that involve only specific columns, use composite indexes.

ASCENDING
Rows are sorted from the smallest value to the largest value. This is the default value.

Alias ASC

Note This option is not valid for Base data sets, SPD Engine data sets, and SPD Server tables.

DESCENDING
Rows are sorted from the largest value to the smallest value.

Alias DESC

Note This option is not valid for Base data sets, SPD Engine data sets, and SPD Server tables.

Details

Overview of Table Indexes

An *index* stores both the values of a table's columns and a system of directions that enable access to rows in that table by index value. Defining an index on a column or set of columns enables SAS, under certain circumstances, to locate rows in a table more quickly and efficiently. Indexes enable SAS to execute the following classes of queries more efficiently:

- comparisons against a column that is indexed
- an IN subquery where the column in the inner subquery is indexed
- correlated subqueries, where the column being compared with the correlated reference is indexed
- join-queries, where the join-expression is an equals comparison and all the columns in the join-expression are indexed in one of the tables being joined

SAS maintains indexes for all changes to the table, whether the changes originate from FedSQL or from some other source. Therefore, if you alter a column's definition or update its values, then the same index continues to be defined for it. However, if an indexed column in a table is dropped, then the index on it is also dropped.

You can create simple or composite indexes. A *simple index* is created on one column in a table. A simple index must have the same name as that column. A *composite index* is one index name that is defined for two or more columns. The columns can be specified in any order, and they can have different data types. A composite index name cannot match the name of any column in the table. If you drop a composite index, then the index is dropped for all the columns that are named in that composite index.

Only the table owner can create an index on a table.

When you create an index, you must know the syntax that the data source supports. Index validation is performed by the data source. FedSQL provides the DBCREATE_INDEX_OPTS= table option to enable you to specify data source-specific index parameters for ODBC databases.

UNIQUE Keyword

The UNIQUE keyword causes SAS to reject any change to a table that would cause more than one row to have the same index value. Unique indexes guarantee that data in one column, or in a composite group of columns, remain unique for every row in a table. A unique index can be defined for a column that includes null or SAS missing values if each row has a unique index value.

See Also

Statements:

- "DROP INDEX Statement" on page 941

Table Options:

- "DBCREATE_INDEX_OPTS= Table Option" on page 1034

CREATE TABLE Statement

Creates a new table.

Category: Data Definition

Restrictions: You cannot overwrite an existing table. You must drop the table by using the DROP TABLE statement and re-create the table with the CREATE TABLE statement.

The full functionality of the CREATE TABLE statement is not supported in FedSQL programs that run in CAS. For CAS information, see "CREATE TABLE Statement" in *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*..

Constraint syntax is defined by the data source. The constraint syntax provided here is a general syntax. For complete constraint syntax, see the documentation for your data source. The SPD Engine and SPD Server data sources do not support integrity constraints.

Data source: SAS data set, SPD Engine data set, SPD Server table, Amazon Redshift, Aster, DB2 under UNIX and PC, Google BigQuery, Greenplum, HAWQ, HDMD, Hive, Impala, JDBC, MDS, Microsoft SQL Server, MongoDB, MySQL, Netezza, ODBC, Oracle, PostgreSQL, Salesforce, SAP HANA, SAP IQ, Spark, Teradata, Vertica, Yellowbrick

Notes: Braces in the syntax convention indicate a syntax grouping. The escape character (\ ) before a brace indicates that the brace is required in the syntax. Table options must be contained by braces ({ }).

MongoDB tables are not relational tables and differ from relational tables in significant ways. In the August 2021 update of SAS/ACCESS on SAS 9.4M7 and SAS Viya 3.5, SAS/ACCESS software adds support for MongoDB structural IDs. See information about

MongoDB in *SAS/ACCESS for Nonrelational Databases: Reference* before creating tables in MongoDB.

When creating a table, Salesforce automatically generates a Name column. If you specify a Name column, the column is ignored. If you must add a column named Name, specify the USE_NATIVE_NAMES= option in the Salesforce connection and qualify the column name as follows: Name_c. Columns in Salesforce tables might not be created in the order in which they were defined. However, the resulting order is preserved in subsequent executions of FedSQL statements on the tables.

SAS 9.4, SAS Viya, and SAS Federation Server can support different data sources. For support information: SAS 9.4 and SAS Viya users, see "Data Source Support" in *Base SAS Procedures Guide*; SAS Federation Server users, see SAS Federation Server Documentation.

Syntax

CREATE TABLE {*table* | _NULL_}
[\{OPTIONS *SAS-table-option=value*
[…*SAS-table-option=value*] \}]
| (<column-definition> [, …<column-definition> | <table-constraint>])
| AS *query-expression*
;

<column-definition>::=
column data-type [<column-constraint>] [DEFAULT *value*]
[HAVING [FORMAT *format*][INFORMAT *informat*][LABEL '*label*']]

<column-constraint>::=
CONSTRAINT *constraint*
{ CHECK (*search-condition*)
| PRIMARY KEY
| UNIQUE
| NOT NULL }

<table-constraint>::=
CONSTRAINT *constraint*
{ CHECK (*search-condition*)
| PRIMARY KEY (*column* [, …*column*])
| UNIQUE (*column* [, …*column*])
| <referential-constraint> }
[<constraint-check-time>]

<referential-constraint>::=
FOREIGN KEY (*referencing-column* [, …*referencing-column*])
REFERENCES *referenced-table* (*referenced-column* [, …*referenced-column*])
[<referential-trigger-action>]

<referential-trigger-action>::=
{ ON UPDATE <referential-action> [ON DELETE <referential-action>] }
| { ON DELETE <referential-action> [ON UPDATE <referential-action>] }

<referential-action>::=

CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION

<constraint-check-time>::=

{ DEFERRABLE [INITIALLY DEFERRED | INITIALLY IMMEDIATE] }
| { [INITIALLY DEFERRED | INITIALLY IMMEDIATE] DEFERRABLE }
| INITIALLY DEFERRED
| { NOT DEFERRABLE [INITIALLY IMMEDIATE] }
| { [INITIALLY IMMEDIATE] NOT DEFERRABLE }

Arguments

table

specifies a name for the table. *table* can be one of these forms

- *catalog.schema.table-name*
- *schema.table-name*
- *catalog.table-name*
- *table-name*

catalog

is an implementation of the ANSI SQL standard for an SQL catalog. The catalog is a data container object that groups logically related schemas. The catalog is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names.

schema

is an implementation of the ANSI SQL standard for an SQL schema The schema is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

table-name

is the name of the table.

Restriction Table names for Hive and HDMD tables should not begin with an underscore (_). In HDFS, a name that starts with _ is a hidden system file. Additionally, table names for HDMD tables should not contain a : (colon), a / (forward slash), or a \ (backslash). They can include a . (period). Table names for Hive tables should not contain a . (period), a : (colon), a / (forward slash), or a \ (backslash).

Requirement Table naming conventions are based on the data source. When more than one data source is involved, the maximum length of a table name is determined by the maximum length that is supported by all of the data sources and FedSQL. For example, if your data sources are a SAS data set that has a maximum of 32 characters and MySQL that has a maximum of 64 characters, the maximum length of a table name is 32 characters. For more information, see the documentation for your data source.

Interaction For data sources that do not support schema, such as SAS data sets, SPD Engine data sets, and SPD Server tables, use the form *catalog.table-name*.

Note Three-level table names are supported in Hive 0.14 and later. When creating or accessing tables in earlier Hive releases, use the two-level form *catalog.table-name*. Support for SQL-standard three-level table names for Hive tables was added in SAS 9.4M4.

NULL
specifies to test the performance of creating a table using the AS *query-expression* clause. FedSQL creates the table internally as if it were to be saved, writing the normal progress messages. The table creation appears to be successful. Once the query expression is complete, the table is discarded.

Interaction A SELECT *select-list* FROM _NULL_ statement returns an error.

{OPTIONS *SAS-table-option=value* [… *SAS-table-option=value*]}
specifies one or more table options and their respective values to apply to the table.

Requirement The OPTION argument and all table options must be enclosed in braces ({ }).

See "FedSQL Statement Table Options" on page 987

AS *query-expression*
specifies to create a new table from an existing table by selecting rows from the existing table using a query expression. The column attributes, such as formats and labels, are copied from the existing table to the new table.

query-expression
specifies the SELECT statement that retrieves information from an existing table to use in creating a new table.

Requirement The number of columns used in the SELECT statement must equal the number of columns in the table.

See "Creating and Populating Tables from a Query Expression" on page 931

"Query Expressions and Subqueries" on page 46

"SELECT Statement" on page 952

Data source SAS data set, SPD Engine data set, SPD Server table, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, HDMD, Hive, JDBC, MDS, Microsoft SQL Server, MySQL, Netezza, ODBC, Oracle, SAP HANA, SAP IQ, Spark, Teradata, Vertica

<column-definition> Arguments

column
specifies the name of the column. The column name is either a local or schema qualified name, using one of these forms:

- *schema.column-name*

- *column-name*

schema
is an implementation of the ANSI SQL standard for an SQL schema. The schema is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

column-name
is the name of the column.

Restrictions For SAS data sets, SPD Engine data sets, and SPD Server tables, the column name cannot be longer than 32 characters.

Hive does not permit a . (period) or a : (colon) within a column name.

Requirement When more than one data source is involved, the maximum length of a column name is determined by the maximum length that is supported by all of the data sources and FedSQL. For example, if your data sources are a SAS data set that has a maximum of 32 characters and MySQL that has a maximum of 64 characters, the maximum length of a column name is 32 characters. Each column in a table must be unique.

See For column name requirements, see the documentation for your data source.

data-type
specifies the type of data that the column can store. If the column definition is for a SAS data set, an SPD Engine data set, or an SPD Server table and the data type is other than CHAR or DOUBLE, FedSQL converts character data types to a CHAR and numeric data types to DOUBLE. FedSQL supports the following table data types:

BIGINT
stores a large signed, exact whole number.

Range -9,223,372,036,854,775,808 to 9,233,372,036,854,775,807

Data source Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, HDMD, Hive, Impala, JDBC, MDS, Microsoft SQL Server, MongoDB, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Spark, Teradata, Vertica, Yellowbrick

Storage size 8 bytes

BINARY(*n*)
stores varying-length binary data.

n
specifies the maximum number of bytes that can be used to store the binary data. The number of bytes that are used to store the binary data is the number of bytes that are necessary to represent the binary data, up to *n* bytes.

Range

Data source DB2 under UNIX and PC, Hive, JDBC, MDS, Microsoft SQL Server, ODBC, SAP HANA, SAP IQ, Spark, Vertica

Storage size up to *n* bytes

CHAR(*n*) [CHARACTER SET "*character-set-identifier*"]
stores a fixed-length character string.

n
specifies the number of characters that are allowed in the character string. If the character string is less than *n* characters, the value is right-padded with spaces. FedSQL supports character semantics for data sources that support character semantics. For SAS data sets, SPD Engine data sets, and SPD Server tables, each character is assigned one byte of storage. For more information, see "Semantics for Character Columns" on page 932.

CHARACTER SET "*character-set-identifier*"
specifies character set encoding information for CHAR data types.

Default

Restriction SAS data sets, SPD Engine data sets, SPD Server tables, Hive, Impala, JDBC, MongoDB, Salesforce, and Spark do not support the use of CHARACTER SET "*character-set-identifier*". You can specify an encoding value for SAS data sets with the "ENCODING= Table Option" on page 1038. Hive, Impala, MongoDB, Salesforce, and Spark character columns are stored as UTF-8.

See "LOCALE= Values for PAPERSIZE and DFLANG, Options" in *SAS National Language Support (NLS): Reference Guide*

Alias CHARACTER(*n*)

Requirements *n* must be specified.

In Salesforce, *n* must be less than or equal to 255 characters. If a value greater than 255 is specified, the column is created as VARCHAR(*n*).

Data source SAS data set, SPD Engine data set, SPD Server table, Amazon Redshift, DB2 under UNIX and PC, Greenplum, HAWQ, Hive, HDMD, Impala, JDBC, MDS, Microsoft SQL Server, MongoDB, MySQL, Netezza, ODBC, Oracle, PostgreSQL, Salesforce, SAP HANA, Spark, Teradata, Vertica, Yellowbrick.

Storage size *n* characters

DATE
stores a date value in the format *yyyy-mm-dd*.

Date Element	Description	Valid Values
yyyy	a four-digit year	0001 – 9999

Date Element	Description	Valid Values
mm	a two-digit month	01 – 12
dd	a two-digit day	00 – 31

Requirement Input values must be specified as a DATE constant. For more information, see "FedSQL Date, Time, and Datetime Constants" on page 59.

Data source SAS data set, SPD Engine data set, SPD Server table, Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, HDMD, Hive, Impala, JDBC, MDS, Microsoft SQL Server, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Salesforce, Spark, Teradata, Vertica, Yellowbrick

Storage size 8 bytes

DECIMAL
stores a signed, fixed-point decimal number.

Alias NUMERIC

Data source Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, HDMD, Hive, JDBC, MDS, Microsoft SQL Server, MySQL, Netezza, ODBC, Oracle, PostgreSQL, Salesforce, SAP HANA, SAP IQ, Spark, Teradata, Vertica, Yellowbrick

Storage size 8 bytes

DOUBLE
stores a signed, approximate, floating point number.

Alias DOUBLE PRECISION

Data source SAS data set, SPD Engine data set, SPD Server table, Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, HDMD, Hive, Impala, JDBC, MDS, Microsoft SQL Server, MongoDB, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Salesforce, Teradata, Vertica, Yellowbrick

Storage size 8 bytes

FLOAT(*p*)
stores a signed, approximate, single-precision or double-precision, floating point number. The user-specified precision determines whether the data type stores a single precision or double precision number. If the specified precision is equal to or greater than 25, the value is stored as a double precision number, which is a DOUBLE. If the specified precision is less than 25, the value is stored as a single precision number, which is a REAL.

p
specifies the maximum number of digits in the floating point number.

Data source SAS data set, SPD Engine data set, SPD Server table, Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, HDMD, Hive, Impala, Microsoft SQL Server, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Spark, Teradata, Vertica, Yellowbrick

Storage size 4 bytes

Tip If *p* is not specified, a DOUBLE is used.

INTEGER

stores an exact whole number.

Range -2,147,483,648 to 2,147,483,647

Data source SAS data set, SPD Engine data set, SPD Server table, Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, HDMD, Hive, Impala, JDBC, MDS, Microsoft SQL Server, MongoDB, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Salesforce, Spark, Teradata, Vertica, Yellowbrick

Storage size 4 bytes

NCHAR(*n*)

stores a fixed-length character string by using the Unicode national character set.

n

specifies the maximum number of multibyte characters that are used to store the character string. If the character string is less than *n* characters, the value is right-padded with spaces.

Requirement *n* must be specified.

Data source Aster, DB2 under UNIX and PC, JDBC, MDS, Microsoft SQL Server, MySQL, ODBC, Oracle, Salesforce, SAP HANA, SAP IQ

Storage size Up to *n* multibyte characters. Depending on the operating system, Unicode characters use either 2 or 4 bytes per character and support all international characters.

NUMERIC

stores a signed, fixed-point decimal number.

Alias DECIMAL

Data source Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, HDMD, Hive, JDBC, MDS, Microsoft SQL Server, MySQL, Netezza, ODBC, Oracle, PostgreSQL, Salesforce, SAP IQ, Spark, Teradata, Vertica, Yellowbrick

Storage size 8 bytes

NVARCHAR(*n*)

stores a varying-length multibyte character string by using the Unicode national character set.

n
specifies the maximum number of multibyte characters that are allowed in the character string. The number of bytes is the actual number of multibyte characters specified, up to *n* characters.

Requirement *n* must be specified.

Data source DB2 under UNIX and PC, JDBC, MDS, Microsoft SQL Server, ODBC, Oracle, Netezza, Salesforce, SAP HANA, SAP IQ, Vertica

Storage size The number of multibyte characters. Depending on the platform, Unicode characters use either two or four bytes per character and can support all international characters.

REAL
stores a signed, approximate, single-precision, floating-point number.

Requirement

Data source SAS data set, SPD Engine data set, SPD Server table, Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, HDMD, Hive, Impala, JDBC, MySQL, Netezza, ODBC, Oracle, PostgreSQL, Salesforce, SAP HANA, SAP IQ, Spark, Vertica, Yellowbrick

Storage size 4 bytes

SMALLINT
stores a small signed, exact whole number.

Range -32,768 to 32,767

Data source SAS data set, SPD Engine data set, SPD Server table, Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, HDMD, Hive, Impala, JDBC, MySQL, Netezza, ODBC, Oracle, PostgreSQL, Salesforce, SAP HANA, SAP IQ, Spark, Teradata, Vertica, Yellowbrick

Storage size 2 bytes

TIME(*p*)
stores a time value with seconds precision.

Time values are in the following form:*hh*:*mm*:*ss*[.*nnnnnn*]

Time Element	Description	Valid Values
hh	a two-digit hour	00 – 23
mm	a two-digit minute	00 – 59
ss	a two-digit second	00 – 61
nnnnnn	up to six digits to indicate a fraction of a second	0 – 999999

p specifies 0–6 digits to use for the precision of a fraction of a second.

Requirement Input values must be specified as a TIME constant. For more information, see "FedSQL Date, Time, and Datetime Constants" on page 59.

Data source SAS data set, SPD Engine data set, SPD Server table, Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, HDMD, Hive, Impala, JDBC, MDS, MySQL, Netezza, Microsoft SQL Server, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Salesforce, Spark, Teradata, Vertica, Yellowbrick

Storage size 8 bytes

TIMESTAMP(*p*)
stores both the date and time value with seconds precision.

Date and time values are in the following form: *yyyy-mm-dd hh*:*mm*:*ss*[.*nnnnnn*]

Time Element	Description	Valid Values
Date Components		
yyyy	a four-digit year	0001- 9999
mm	a two-digit month	01- 12
dd	a two-digit day	01- 31
Time Components		
hh	a two-digit hour	00 – 23
mm	a two-digit minute	00 – 59
ss	a two-digit second	00 – 61
nnnnnn	up to six digits to indicate a fraction of a second	0 – 999999

p specifies 0–6 digits to use for the precision of a fraction of a second.

Alias DATETIME

Requirement Input values must be specified as a TIMESTAMP constant. For more information, see "FedSQL Date, Time, and Datetime Constants" on page 59.

Data source SAS data set, SPD Engine data set, SPD Server table, Amazon Redshift, DB2 under UNIX and PC, Greenplum,

HAWQ, HDMD, Hive, Impala, JDBC, MDS, Microsoft SQL Server, MongoDB, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Salesforce, Spark, Teradata, Vertica, Yellowbrick

Storage size 8 bytes

TINYINT

stores a very small signed exact, whole number.

Range -128 to 127

Data source SAS data set, SPD Engine data set, SPD Server table, HDMD, Hive, Impala, JDBC, Microsoft SQL Server, MySQL, Netezza, ODBC, Oracle, Salesforce, SAP HANA, SAP IQ, Spark, Teradata, Vertica

Storage size 1 byte

VARBINARY(*n*)

stores varying-length binary data.

n

specifies the maximum number of bytes that are allowed in the binary string. The number of bytes that are used to store the binary data is the number of bytes that are necessary to represent the binary data, up to *n* bytes.

Range

Data source DB2 under UNIX and PC, Hive, JDBC, MDS, Microsoft SQL Server, ODBC, Oracle, SAP HANA, SAP IQ, Spark, Teradata, Vertica

Storage size up to *n* bytes

VARCHAR(*n*) [CHARACTER SET "*character-set-identifier*"]

stores a varying-length character string.

n

specifies the number of characters that are allowed in the character string. The number of characters that are stored is the actual number of characters, up to *n* characters. FedSQL supports character semantics for data sources that support character semantics. For SAS data sets, SPD Engine data sets, and SPD Server tables, each character is assigned one byte of storage. For more information, see “Semantics for Character Columns” on page 932.

[CHARACTER SET "*character-set-identifier*"]

specifies character set encoding information for CHAR data types.

Default The default encoding depends on your operating system and locale.

Restriction Hive, JDBC, MongoDB, Salesforce, and Spark do not support the use of CHARACTER SET "*character-set-identifier*". Hive, JDBC, MongoDB, Salesforce, and Spark VARCHAR columns are stored as UTF-8. See Table A2.11

on page 1126 for more information about Hive support for VARCHAR columns.

See "LOCALE= Values for PAPERSIZE and DFLANG, Options" in *SAS National Language Support (NLS): Reference Guide*

Data source Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, HDMD, Hive, Impala, JDBC, MDS, Microsoft SQL Server, MongoDB, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Salesforce, Spark, Teradata, Vertica, Yellowbrick

Storage size up to *n* characters

<column-constraint>
specifies to place an integrity constraint on the column.

Data source SAS data set, DB2 under UNIX and PC, MDS, Microsoft SQL Server, MongoDB, MySQL, ODBC, Oracle, Salesforce, SAP HANA, Teradata, Vertica

See "<column-constraint> and <table-constraint> Arguments" on page 927

"Integrity Constraints" on page 932

DEFAULT *value*
for each row that is added to the table, specifies a value for the column that is assumed when no other value is entered for that column.

value
specifies the default value.

Requirement *value* must be in the range of the specified data type.

Data source DB2 under UNIX and PC, Microsoft SQL Server, MySQL, ODBC, Oracle, SAP HANA, Teradata

HAVING [FORMAT *format*][INFORMAT *informat*][LABEL '*label*']
specifies a clause that is used to associate a format, informat, or label with the column.

FORMAT *format*
specifies a SAS data format that is stored as column metadata. The format is not applied to the column data until execution time.

format
specifies a valid SAS data format. If the format is a format other than a FedSQL or DS2 format, it must be a valid format for the Base SAS.

See "FedSQL Formats" on page 75

"DS2 Formats" in *SAS DS2 Language Reference*

"Dictionary of Formats" in *SAS Formats and Informats: Reference*

Restriction Formats that are created by PROC FORMAT cannot be stored as column metadata.

Note When you create a table, you can associate a SAS data format with a column. The format is not validated or applied to the column until execution time, such as in a PUT function. If the format is applied to the column in a Base SAS environment, any format can be stored with the column. If the format is applied to the column in the FedSQL environment, the format must be a valid FedSQL or DS2 format.

INFORMAT *informat*
specifies a SAS data informat that is stored as column metadata. The informat is not applied to the column data. No validation is done on the informat; it is only informational.

informat
specifies a SAS data informat.

See Chapter 6, “FedSQL Informats,” on page 899

“Dictionary of Informats” in *SAS Formats and Informats: Reference*

LABEL '*text-string*'
specifies a text string to use as an alternate column heading that appears in place of the column name in a query expression result.

Range 1 - 255 characters

Restriction A label can be created only with the CREATE TABLE statement and it cannot be used in a query expression operation.

Requirement *text-string* must be enclosed in either double or single quotation marks. If *text-string* contains a single quotation mark, enclose *text-string* in double quotation marks.

Data source SAS data set, SPD Engine data set, SPD Server table

<column-constraint> and <table-constraint> Arguments

CONSTRAINT
begins a column constraint or a table constraint.

Data source SAS data set, DB2 under UNIX and PC, MDS, Microsoft SQL Server, MongoDB, MySQL, ODBC, Oracle, SAP HANA, Teradata, Vertica

constraint
specifies a name to identify the constraint.

See For constraint name requirements, see the documentation for your data source.

CHECK(*search-condition*)
specifies a condition for values in the column or table. *search-condition* is a valid FedSQL expression that resolves to a Boolean value. If *search-condition* is false, no changes are made to the table.

Restrictions For a SAS data set, the search condition cannot contain FedSQL functions.

The CHECK constraint is supported on Vertica 7.2 and later.

Data source SAS data set, Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, Microsoft SQL Server, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Teradata, Vertica, Yellowbrick

See "FedSQL Expressions" on page 46

PRIMARY KEY
specifies that, for each row in the table, the value in the column can be used to uniquely identify its respective row in the table. It is a value that does not change. A primary key cannot have a null value.

Data source SAS data set, Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, Microsoft SQL Server, MongoDB, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Teradata, Vertica

Note See *SAS/ACCESS for Nonrelational Databases: Reference* for information about how PRIMARY KEY constraints are used in MongoDB.

PRIMARY KEY (*column* [, …*column*])
specifies that, for each row in the table, the values for all of the columns specified are used to uniquely identify its respective row in the table. A primary key cannot have a null value and it must be unique for each row in the column. It is a value that does not change.

Data source SAS data set, Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, Microsoft SQL Server, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Teradata, Vertica

UNIQUE
specifies that no two rows in the column can have the same value. A null value is allowed unless you specify NOT NULL.

Data source SAS data set, Amazon Redshift, DB2 under UNIX and PC, MDS, Microsoft SQL Server, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, Teradata, Vertica, Yellowbrick

Note For Vertica, see the vendor documentation for information about how the UNIQUE constraint is enforced.

UNIQUE(*column* [, …*column*])
specifies that no two rows in any of the specified columns can have the same value. A null value is allowed unless you specify NOT NULL.

Data source SAS data set, Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, MDS, Microsoft SQL Server, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Teradata, Vertica, Yellowbrick

NOT NULL
specifies that a null value is not valid in any row for the specified column.

Data source SAS data set, Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, Hive, MDS, Microsoft SQL Server, MySQL, ODBC, Oracle, PostgreSQL, Salesforce, SAP HANA, SAP IQ, Teradata, Vertica, Yellowbrick

FOREIGN KEY (*referencing-column* [, …*referencing column*]) REFERENCES *referenced-table* (*referenced-column* [, …*referenced-column* <referential-trigger-action>])

specifies the clause that relates columns in *table* to columns in another table through the values for those columns. Foreign keys must always include enough columns to uniquely identify a row in the referenced table. Foreign key constraints help ensure the integrity of related data in multiple tables.

FOREIGN KEY

begins the clause that identifies one or more columns in *table* that relate to columns in another table.

Data source Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, HAWQ, Microsoft SQL Server, MongoDB, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Teradata, Vertica

Note In MongoDB, a FOREIGN KEY constraint is used to define a relationship with a parent table. Referential triggers, actions, and check times are not supported by MongoDB. See *SAS/ACCESS for Nonrelational Databases: Reference* for more information about how referential relationships are used in MongoDB.

referencing-colunm

specifies the columns in *table* that relate to columns in another table.

REFERENCES

begins the clause that identifies the columns in another table.

referenced-table

specifies the table whose columns relate to columns in *table*.

referenced-column

specifies one or more columns in the referenced table that relate to the columns in *table*.

<referential-trigger-action>

specifies the action to take in the referenced table when columns in *table* are updated or deleted.

ON UPDATE <referential-action>

specifies the clause for the action to take when a column in *table* is updated.

ON DELETE <referential-action>

specifies the clause for the action to take when a column in *table* is deleted.

<referential-action>

specifies the action to take when a column in *table* is updated or deleted.

CASCADE

ON UPDATE, specifies to update all rows in *table* that reference an updated value in the referenced table. ON DELETE, removes all rows in *table* that reference the deleted rows in the referenced table.

SET NULL

specifies to set the referenced columns to a null value.

SET DEFAULT
specifies to set the referenced columns to their default value.

RESTRICT
specifies that if the referential constraint is not satisfied at any time while the FedSQL statement is processing, no action is taken and the update or deletion fails.

NO ACTION
specifies that if the referential constraint is not satisfied at the end of the FedSQL statement, no action is taken and the update or deletion fails.

<constraint-check-time>
specifies when the constraint is checked:

DEFERRABLE
NOT DEFERRABLE
specifies whether the constraint violation check can be performed after the transaction completes or if it must be checked at the end of each FedSQL statement. If DEFERRABLE is specified, the constraint check can occur at the end of the transaction. If NOT DEFERRABLE is specified, the constraint must be checked when the FedSQL statement terminates. Specifying DEFERRABLE is useful when more than one statement is necessary to complete a transaction.

Default DEFERRABLE

INITIALLY DEFERRED
specifies that the constraint violation check is deferred, by default, at the beginning of each transaction and it does not occur until the end of a transaction.

INITIALLY IMMEDIATE
specifies that the constraint violation check occurs at the end of each FedSQL statement.

Details

Overview of the CREATE TABLE Statement

The CREATE TABLE statement enables you to create tables by defining table columns or by selecting columns from an existing table using a query expression. SAS provides extensions to the CREATE TABLE statement to support SAS data sources.

Defining Columns for a Table

When you define columns for a table, the column name and the data type are required. All other column definition arguments are optional. For column naming conventions, see the documentation for your data source. FedSQL reserved words cannot be used as column names. For a list of FedSQL reserved words, see “FedSQL Reserved Words” on page 71 .

You can add an unspecified number of columns to a table by separating each column definition with a comma. Enclose the complete group of column definitions

in parenthesis. The following CREATE TABLE statement creates the Customers table:

```
create table customers
   ( custid double unique,
     name char(16),
     address char(64),
     city char(16),
     state char(2),
     country char(16),
     phone char(16),
     initorder date having label 'Initial Order'
   );
```

Each column definition above provides the name and assigns the data type for that column. The data types that are supported for data definition depend on the target data source. The semantics for character columns and availability of integrity constraints also depend on the data source.

The preceding example shows how to create a Base SAS data set with FedSQL. The CUSTID column contains the UNIQUE integrity constraint, which ensures that column values are unique (there are no duplicated values). The INITORDER column is of the DATE data type. The Base engine does not support the DATE data type. FedSQL supports the DATE type for data definition for portability with other data sources, and internally creates the column as type DOUBLE with a DATE9. format for storage. The label `Initial Order` is a SAS extension that is supported for SAS compatibility. When the column is displayed, the label `Initial Order` is used instead of the column name `initorder`. The LABEL modifier is ignored for non-SAS data sources. For more information about this and other column modifiers to support compatibility with SAS, see "SAS Extensions for the CREATE TABLE Statement to Support SAS Data Sources" on page 933.

You can add rows to the table by using the INSERT statement. The INSERT statement enables you to specify values for each of the table columns defined with the CREATE TABLE statement. For more information, see the "INSERT Statement" on page 948.

Creating and Populating Tables from a Query Expression

When you create a table using a query expression, you add rows to the table as the table is created. You use a SELECT statement to retrieve data from an existing table to create the new table. You can specify the names of the columns that you want to retrieve in the CREATE TABLE statement, or you can specify a * (asterisk) to retrieve all columns from the source table.

This CREATE TABLE statement creates a new table that is based on only three columns from the CorpData table:

```
create table spainEmails
as select name, emailid, lastPurchaseDate from corpdata where country='Spain';
```

The following CREATE TABLE statement selects all columns from the CorpData table:

```
create table spain
as select * from corpdata where country='Spain';
```

You can test the performance of a query expression before creating a table by using a table name of _NULL _ for the query expression, as in this example:

```
create table _null_ as select * from corpdata where country='Spain';
```

FedSQL performs normal processing to create the table and the table appears to be created. The table is discarded when the query is complete.

See "FedSQL Expressions" on page 46 for examples of other query expressions.

When creating a new SAS data set, SPD Engine data set, or SPD Server table from an existing SAS data set, SPD Engine data set, or SPD Server table, FedSQL preserves any existing formats and labels in the new table. It does not support definition of formats and labels when creating a table with the AS expression.

Data Sources

When defining data types using the FedSQL language, in order for data to be stored, the defined data type must be available for data storage in that data source. Although FedSQL provides support for several data types, the data types that can be defined for a particular table depend on the data source. Each data source does not necessarily support all of the FedSQL data types.

In addition, data sources support variations of the standard SQL data types. That means that a data type that you specify might default to a different data type that might also have different attributes in the underlying data source. This is done when a data source does not natively support a specific data type, but data values of a similar data type can be converted without data loss. For example, to support the INTEGER data type, a SAS data set defaults the data type definition to SAS numeric, which is a DOUBLE. For details about data source implementation for each data type such as data source-dependent attributes, see Appendix 2, "Data Type Reference," on page 1109.

Semantics for Character Columns

FedSQL supports character semantics for data sources that support character semantics. That is, when defining a character column for Oracle or another DBMS that supports character semantics, *n* in CHAR(*n*) is the number of characters, regardless of whether the character is from a single-byte character set or from a multibyte character set. The data source automatically adjusts the storage size to the encoding. For data sources that support byte semantics, such as Base SAS, SPD Engine, and SPD Server, FedSQL allocates one byte per character for storage. You must adjust *n* to account for the extra bytes used by a multibyte encoding. Otherwise, the data might be truncated. Characters from multibyte character sets typically require two to four bytes of storage.

Integrity Constraints

Integrity constraints are a set of data validation rules that you can specify to preserve the validity and consistency of your data. When a transaction modifies the table, the data source enforces integrity constraints. Integrity constraints that are specified in the CREATE TABLE statement are passed through to the data source for creation and enforcement. If the data source does not support the integrity constraint, FedSQL returns an error.

You can define column and table constraints. A column constraint is a constraint that is defined for one column. A table constraint defines a constraint for two or more columns.

The following statements create SAS data sets using integrity constraints:

- Create a Products table using the product ID as the primary key; no two product IDs can identify the same product. Also, the product name cannot be a null value. Check constraints at the end of a transaction.

```
create table products (prodid double primary key,
   product char(8) unique not null initially deferred);
```

- Create a Sales table. The CustID value must be unique. The Country value cannot be a null value.

```
create table sales (prodid double,
   custid double  primary key,
   totals double having format dollar10.,
   country char(30) not null);
```

- Create a SAS data set for customer credit card information. Customers are uniquely identified by their name and customer number.

```
create table custcredit (name char(30),
   custNum double,
   ccType char(15) having label 'Credit Card Type',
   ccNum char(20) having label 'Credit Card Number',
   ccExp date having label 'Expiration Date',
   CONSTRAINT ccconst primary key(name, CustNum));
```

For information about constraints, see the documentation for your data source.

SAS Extensions for the CREATE TABLE Statement to Support SAS Data Sources

SAS extensions for the CREATE TABLE statement using FedSQL enable you to assign formats and informats to columns and add a label to a column.

SAS Formats

You can specify that the column data is stored and retrieved as a SAS formatted value by using the HAVING FORMAT clause. If the data type is not either CHAR or DOUBLE, it is converted to either CHAR or DOUBLE.

For example, to store the date in the format *ddmmmyy*, where *dd* is a two-digit year, *mmm* is a three-digit month, and *yy* is a two-digit year, the column definition would specify DOUBLE as the data type and FORMAT='DATE7.'. November 1, 2012 would be stored as 01Nov12. Your column definition might look like the following:

```
saleenddate double not null having format date7.;
```

In comparison, a column that stores date information using the DATE data type stores the date in the format *yyyy-mm-dd*. November 1, 2012 would be stored as 2012-11-01.

SAS Informats

SAS and user-defined informats can be stored and retrieved with the column data by using the HAVING INFORMAT clause. The informat is not applied to the data; it is information only. The client application is responsible for applying the

informat to the data. Informats can be associated with all data types, but all data types are converted to either CHAR or DOUBLE.

```
saledates double having informat anydtdte14.;
```

For more information about SAS informats, see Chapter 6, “FedSQL Informats,” on page 899.

Column Labels
A label is a descriptive, quoted, text string that is displayed in query expression results instead of the column name. You specify a label by using the LABEL argument in the HAVING clause:

```
saleenddate double not null having label 'Last Day of Sale';
```

A label cannot be used in a query expression operation.

See Also

Statements

- “ALTER TABLE Statement” on page 905
- “DROP TABLE Statement” on page 942

CREATE VIEW Statement

Creates a FedSQL view of data from one or more tables or other views.

Category:	Data Definition
Restriction:	This statement is not supported in FedSQL programs that run in CAS.
Data source:	SAS data set, SPD Server table, Amazon Redshift, Aster, DB2 under UNIX and PC, Google BigQuery, Greenplum, HAWQ, HDMD, Hive, Impala, JDBC, MDS, Microsoft SQL Server, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Spark, Teradata, Vertica, Yellowbrick
Note:	SAS 9.4, SAS Viya, and SAS Federation Server can support different data sources. For support information: SAS 9.4 and SAS Viya users, see “Data Source Support” in *Base SAS Procedures Guide*; SAS Federation Server users, see SAS Federation Server Documentation.

Syntax

CREATE VIEW *view* [SECURITY *security-type*] AS *query-expression*;

Arguments

SECURITY *security-type*
specifies the type of security that will be enforced for the view. *security-type* can be one of the following values.

DEFINER
specifies that the view is run with the schema owner's credentials.

INVOKER
specifies that the view is run with the invoking user's credentials.

Default If SECURITY is not used, security for the view defaults to INVOKER.

Restriction Security is available only with SAS Federation Server. The SECURITY *security-type* argument is ignored in a FedSQL request that is not directed to SAS Federation Server. For more information about the security available through SAS Federation Server, see the documentation for SAS Federation Server.

view
specifies the name of the view being created.

Requirement The view name must be different from any other view, table, or index in the same database.

query-expression
specifies the SELECT statement that retrieves the information from an existing table that is used to create the view.

See "Creating and Populating Tables from a Query Expression" on page 931, "Query Expressions and Subqueries" on page 46, and "SELECT Statement" on page 952

Details

Overview of the CREATE VIEW Statement

A view that is created with the FedSQL CREATE VIEW statement is not a DBMS view or a SAS view. The output of the CREATE VIEW statement is a native table that is viewable only with the FedSQL language.

If an underlying table or view in a view is deleted, SAS returns an error message when the view is executed. If the underlying table or view is changed, the view might have to be re-created.

Access to Views

To create a view, you must have CREATE VIEW privilege. As the view owner, you have all privileges for the view, including the ability to grant privileges to other users. To execute a view, the user must have SELECT privilege on the tables and views that are referenced in the view.

Privileges are currently supported only in SAS Federation Server.

SELECT Statement Restrictions

The SELECT statement cannot include an ORDER BY clause.

FedSQL Views and SAS

The SAS procedures CONTENTS, PRINT, and SQL cannot access the data in a FedSQL view. However, these procedures print information about the structure of a FedSQL view.

See Also

Statements:

- "CREATE TABLE Statement" on page 915
- "DESCRIBE VIEW Statement" on page 940
- "DROP VIEW Statement" on page 944
- "SELECT Statement" on page 952

DELETE Statement

Deletes rows from a table.

Category:	Data Manipulation
Restriction:	This statement is not supported in FedSQL programs that run in CAS.
Data source:	SAS data set, SPD Engine data set, SPD Server table, Amazon Redshift, Aster, DB2 under UNIX and PC, Google BigQuery, Greenplum, JDBC, MDS, Microsoft SQL Server, MongoDB, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Teradata, Vertica, Yellowbrick
Note:	SAS 9.4, SAS Viya, and SAS Federation Server can support different data sources. For support information: SAS 9.4 and SAS Viya users, see "Data Source Support" in *Base SAS Procedures Guide*; SAS Federation Server users, see SAS Federation Server Documentation.

Syntax

DELETE FROM *table* [WHERE <search-condition>];

Arguments

table

specifies the table from which you are deleting rows.

Restriction If row-level permissions are in effect for the table, you cannot delete rows from the table. Row-level security is available only with SAS Federation Server. See the SAS Federation Server documentation for more information.

Tip You can use '.*' after the table name for compatibility with Microsoft Access tables.

WHERE <search-condition>
specifies any valid WHERE clause used to limit the number of rows that are deleted.

Note Only single variable operators are allowed in MongoDB subqueries. FedSQL does not support the use of subqueries as inner queries to delete data in MongoDB child tables or metadata tables.

Tip You can use parameter arrays in the WHERE statement.

See WHERE clause in the "SELECT Statement" on page 952

Details

The DELETE statement deletes the rows that satisfy the WHERE clause from the specified table. If you do not use a WHERE clause, all rows are deleted.

Comparisons

The DROP TABLE statement removes the table from the database completely, including the table structure. The DELETE statement deletes only the rows (data) in a table.

See Also

Statements:

- "DROP TABLE Statement" on page 942
- "INSERT Statement" on page 948
- "UPDATE Statement" on page 984

DESCRIBE TABLE Statement

Retrieves SQL from a table and returns a result set.

Categories:	Data Definition Data Retrieval
Restriction:	This statement is not supported in FedSQL programs that run in CAS.
Data source:	All

Syntax

DESCRIBE TABLE [*catalog*.][*schema*.] *table*;

Arguments

catalog
specifies the catalog that contains the table.

schema
specifies the schema that contains the table.

table
specifies the name of the table.

Details

The DESCRIBE TABLE statement writes a CREATE TABLE statement to the SAS log for the specified table, regardless of how the table was originally created (for example, with another programming language). The column definitions returned by the DESCRIBE TABLE statement show the column's native data type. If indexes are defined on columns in the table, then CREATE INDEX statements for those indexes are written to the SAS log. The CREATE TABLE statement also includes any integrity constraints that are defined for the specified table. For SAS data sets, primary key integrity constraints are described by a CREATE UNIQUE INDEX statement. Foreign key variables that reference a primary key constraint are not displayed as part of the primary key constraint's DESCRIBE TABLE output.

The FedSQL language supports most of the data types used by the data sources that it supports, but not all. Refer to Appendix 2, "Data Type Reference," on page 1109 for a listing of the data types that FedSQL supports for each data source. If you have a need to create a table with a data type that FedSQL does not support directly, use the EXECUTE statement to make the request.

The DESCRIBE TABLE statement is not supported for FedSQL DICTIONARY tables.

Examples

Example 1: Information Returned for a SAS Data Set

Consider a SAS data set created with the SAS DATA step:

```
data myfiles.employees;
    input IdNum $4. +2 LName $11. FName $11. JobCode $3.
          +1 Salary 5. +1 Phone $12.;
```

The DESCRIBE TABLE statement writes the following information to the SAS log:

```
describe table myfiles.employees;
```

```
CREATE TABLE MYFILES."MYFILES".EMPLOYEES (
     "IdNum"     CHARACTER(4),
     "LName"     CHARACTER(11),
     "FName"     CHARACTER(11),
     "JobCode"     CHARACTER(3),
     "Salary"     DOUBLE,
     "Phone"     CHARACTER(12)
);
```

Example 2: Information Returned for a Supported Data Type

Consider an Oracle table created by using FedSQL with the following statement:

```
CREATE TABLE ORACLE.FOO1 (COL1 INTEGER, COL2 VARHCHAR(10);
```

Oracle does not support the INTEGER data type. The FedSQL language converts the INTEGER data type to the Oracle NUMBER data type when writing data to Oracle.

```
DESCRIBE TABLE oracle.foo1;
```

```
CREATE TABLE "ORACLE"."SCOTT".FOO1 (
     COL1     NUMBER,
     COL2     VARCHAR2(10))
```

Example 3: Information Returned for a Data Type for Which FedSQL Has Limited Support

Consider a DB2 table created with the following statement by using the DB2 driver natively:

```
CREATE TABLE dlfoo1 (col1 CLOB);
```

FedSQL has limited support for the CLOB data type. It does not currently allow creation of columns of the CLOB data type, but it does have some support for reading CLOB columns. The DESCRIBE TABLE statement writes the following information to the SAS log:

```
DESCRIBE TABLE dlfoo1;
```

```
CREATE TABLE "DB2"."DBITEST".DLFOO1 (
     COL1     CLOB
);
```

Example 4: Information Returned for Teradata

Consider a Teradata table created by using FedSQL:

```
create table x.customer (item_name char(30), cost decimal(10,2), delivered date);
```

The DESCRIBE TABLE statement returns a CREATE TABLE statement that describes the columns in the table. It also returns a CREATE INDEX statement, although an index was not explicitly defined. The index is a FedSQL abstraction. Do not use it to re-create the table.

```
describe table x.customer;
```

```
CREATE TABLE X."dmvdq1".CUSTOMER (
     ITEM_NAME     CHAR(60) DEFAULT NULL,
     COST     DECIMAL DEFAULT NULL,
     DELIVERED     DATE DEFAULT NULL
);
CREATE INDEX NONAME1 ON X."dmvdq1".CUSTOMER(ITEM_NAME);
```

See Also

Statements:

- "CREATE TABLE Statement" on page 915

DESCRIBE VIEW Statement

Retrieves SQL from a view and returns a result set.

Categories:	Data Definition Data Retrieval
Restrictions:	This statement is not supported in FedSQL programs that run in CAS. This statement is not supported for SAS data views created with the SAS DATA step or PROC SQL.
Data source:	SQL-based data sources that support views.

Syntax

DESCRIBE VIEW ["*catalog*".]["*schema*".]"*view*"
DESCRIBE VIEW XML ["*catalog*".]["*schema*".]"*view*"

Arguments

catalog
specifies the catalog that contains the view.

schema
specifies the schema that contains the view.

view
specifies the name of the view.

Details

Use DESCRIBE VIEW to retrieve standard SQL text from a FedSQL or DBMS view. Use DESCRIBE VIEW XML to retrieve SQL from a view as XML. This eliminates the need to implement an SQL parser on the client.

You must have ALTER VIEW privilege to run DESCRIBE VIEW on a secured view.

See Also

Statements:

- "CREATE VIEW Statement" on page 934

DROP INDEX Statement

Removes the specified index from a table.

Category:	Data Definition
Restriction:	This statement is not supported in FedSQL programs that run in CAS.
Data source:	SAS data set, SPD Engine data set, SPD Server table, Amazon Redshift, Aster, DB2 under UNIX and PC, Greenplum, JDBC, MDS, Microsoft SQL Server, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Teradata
Note:	SAS 9.4, SAS Viya, and SAS Federation Server can support different data sources. For support information: SAS 9.4 and SAS Viya users, see "Data Source Support" in *Base SAS Procedures Guide*; SAS Federation Server users, see SAS Federation Server Documentation.

Syntax

DROP INDEX *index* [FROM *table*] [FORCE];

Arguments

index
specifies the name of the index to be removed.

FROM *table*
specifies the name of the table where the index resides.

Requirements For a SAS data set, SPD Engine data set, SPD Server table, and MDS table, you must include the FROM *table* syntax in order to specify the name of the table where the index resides.

For all data sources, you must include the FROM *table* syntax if you connected using a DSN that is secured with SAS security.

FORCE
specifies that the index is removed without error processing. Use the FORCE keyword only when you are certain that removing the index without error processing will not negatively affect the table.

Details

If you drop a composite index, then the index is removed for all the columns that are named in that index.

Note: The DROP INDEX statement does not apply to indexes that were created by using the PRIMARY KEY or UNIQUE constraints of either the CREATE TABLE or ALTER TABLE statements, respectively. To remove those indexes, you must first remove the constraint by using the ALTER TABLE statement.

See Also

Statements:

- "ALTER TABLE Statement" on page 905
- "CREATE INDEX Statement" on page 913

DROP TABLE Statement

Removes a table from the database.

Category: Data Definition

Restriction: The DROP TABLE statement behaves differently in CAS. For CAS information, see "DROP TABLE Statement" in *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*.

Data source: SAS data set, SPD Engine data set, SPD Server table, Amazon Redshift, Aster, DB2 under UNIX and PC, Google BigQuery, Greenplum, HAWQ, HDMD, Hive, Impala, JDBC, MDS, Microsoft SQL Server, MongoDB, MySQL, Netezza, ODBC, Oracle, PostgreSQL, Salesforce, SAP HANA, SAP IQ, Spark, Teradata, Vertica, Yellowbrick

Notes: Drop table operations are not supported for MongoDB child tables that are created from structural IDs. For more information, see documentation for MongoDB in *SAS/ACCESS for Nonrelational Databases: Reference*.

SAS 9.4, SAS Viya, and SAS Federation Server can support different data sources. For support information: SAS 9.4 and SAS Viya users, see "Data Source Support" in *Base SAS Procedures Guide*; SAS Federation Server users, see SAS Federation Server Documentation.

Syntax

DROP TABLE *table* [FORCE];

Arguments

table
specifies the name of the table to be removed.

Restrictions You cannot alter or drop an MDS table while it is referenced in another transaction or statement. If a request fails, make sure other users are no longer using the table or have disconnected.

You cannot drop a MongoDB table that has a child table. To remove a MongoDB table that has a child table, you must drop the child table first.

FORCE
specifies that the table is dropped without error processing.

Details

The DROP TABLE statement removes a table definition and all the table's data, metadata, and indexes.

If you use the DELETE statement to remove all the rows in a table, the table still exists until it is removed with the DROP TABLE statement.

You cannot use the DROP TABLE statement to remove a table that is referenced by a foreign key constraint. You must drop the foreign key constraint first, and then remove the table.

If you remove a table with indexed columns, then all the indexes are automatically removed. If you drop a composite index, then the index is removed for all the columns that are named in that index.

When specifying the DROP TABLE statement in the same submission as a CREATE TABLE statement, specify the FORCE option in the DROP TABLE statement. The DROP TABLE statement writes messages to the log when the specified table cannot be found. These messages can interfere with the CREATE TABLE statement. The FORCE option suppresses these messages so that the CREATE TABLE request can succeed.

Teradata users: See Appendix 8, “Usage Notes,” on page 1231.

Comparisons

The DELETE statement deletes only the rows (data) in a table. The DROP TABLE statement removes the table from the database completely, including the table structure.

Examples

Example 1: Drop a Table

```
proc fedsql;
drop table mytable;
quit;
```

Example 2: Drop a Table and Create a Table at the Same Time

```
proc fedsql;
drop table mytable force;
create table mytable (col1 integer, col2 char(5));
quit;
```

See Also

Statements:

- "CREATE TABLE Statement" on page 915
- "DELETE Statement" on page 936

DROP VIEW Statement

Removes a view from the database.

Category:	Data Definition
Restriction:	This statement is not supported in FedSQL programs that run in CAS.
Data source:	SAS data set, SPD Server table, Amazon Redshift, Aster, DB2 under UNIX and PC, Google BigQuery, Greenplum, HAWQ, HDMD, Hive, Impala, JDBC, MDS, Microsoft SQL Server, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Spark, Teradata, Vertica, Yellowbrick
Note:	SAS 9.4, SAS Viya, and SAS Federation Server can support different data sources. For support information: SAS 9.4 and SAS Viya users, see "Data Source Support" in *Base SAS Procedures Guide*; SAS Federation Server users, see SAS Federation Server Documentation.

Syntax

DROP VIEW *view* [FORCE];

Arguments

view
specifies the name of the view to be removed.

FORCE
specifies that the view is removed without error processing.

Details

A view can be considered a virtual table. The view is formed by running a query expression against one or more tables. Dropping a view does not change any data in the database. Only the metadata that is associated with the view is deleted.

Any view on a dropped table can be dropped explicitly by using the DROP VIEW statement.

Comparisons

The DROP TABLE statement removes a table from the database. The DROP VIEW statement removes a view from the database.

See Also

Statements:

- "CREATE VIEW Statement" on page 934
- "DESCRIBE VIEW Statement" on page 940
- "DROP TABLE Statement" on page 942

EXECUTE Statement

Sends a DBMS-specific statement to a DBMS that FedSQL supports.

Category:	FedSQL pass-through
Restriction:	This statement is not supported in FedSQL programs that run in CAS.
Data source:	Amazon Redshift, Aster, DB2 under UNIX and PC, Google BigQuery, Greenplum, HAWQ, HDMD, Hive, JDBC, MySQL, Microsoft SQL Server, MongoDB, Netezza, ODBC, Oracle, PostgreSQL, Salesforce, SAP HANA, SAP IQ, Spark, Teradata, Vertica, Yellowbrick
Note:	SAS 9.4, SAS Viya, and SAS Federation Server can support different data sources. For support information: SAS 9.4 and SAS Viya users, see "Data Source Support" in *Base SAS Procedures Guide*; SAS Federation Server users, see SAS Federation Server Documentation.

See: "Data Source Connection" on page 6
"FedSQL Explicit Pass-Through Facility" on page 69
SQL documentation for your DBMS

Syntax

EXECUTE(*native-syntax*) **BY** *catalog*;

Arguments

native-syntax
specifies a native query that can be run on the catalog's driver. The EXECUTE statement accepts statements that produce a result set, as well as statements that do not produce a result set.

Restriction PROC FEDSQL does not support native syntax that contains embedded bare semicolons. You can, however, use the FedSQL language processor to run these statements.

Note The SQL statement might be case-sensitive, depending on your data source, and it is passed to the data source exactly as you enter it.

catalog
specifies the name of a catalog in the existing FedSQL connection.

Details

General Use

The EXECUTE statement is the only way that FedSQL provides to submit native SQL statements that do not produce a result set to a DBMS. Native SQL can be used in the following statements: DDLs, UPDATE, SELECT, and bulk loads.

Use with HDMD

A native connection for HDMD is a FedSQL connection.

Use with MongoDB

MongoDB is not an SQL-based data source. In order to read and write data from MongoDB, SAS/ACCESS software generates a relational schema. By default, this schema exists in memory, but the schema can be stored. You can use the EXECUTE statement to submit a REFRESH command to generate and overwrite a stored MongoDB schema with a new schema.

Use this command to update a stored schema only when database changes, such as additions and deletions of collections or columns, are made by a means other than SAS. When you add new collections with the CREATE TABLE statement or delete collections with the DROP TABLE statement, do not use this command. Structural changes made with SAS are instantly saved to the schema when the SCHEMA_COLLECTION= option is set in the database connection.

The REFRESH command is submitted from a SAS session as follows:

```
proc fedsql;
   execute (refresh) by libref;
quit;
```

Execution of the REFRESH command is an administrative task and should be performed only by the system administrator. REFRESH forces a scan of the current MongoDB database collections and updates the current schema (whether in memory or saved by name) accordingly. This overwrites any manual changes that might have been made directly to a stored schema, so this command should be used with care.

For more information about MongoDB schemas, see “Working with Schemas for MongoDB Data” in *SAS/ACCESS for Nonrelational Databases: Reference*.

You can also use FedSQL explicit pass-through to submit JSON-style MongoDB commands to query, create, and update MongoDB collections. For supported syntax, see information about MongoDB in *SAS/ACCESS for Nonrelational Databases: Reference*.

Examples

Example 1: Using EXECUTE to Create and Drop a Table in SAP HANA

```
execute(create table t_blob (col1 blob)) by saphana;
execute(drop table t_blob) by saphana;
```

Example 2: Using EXECUTE to Produce a Result Set in Oracle

```
execute(select i, rank() over (order by j) rank from table_a ) by
oracle;
```

See Also

Concepts:

- “FedSQL Explicit Pass-Through Facility” on page 69

Statements:

INSERT Statement

Adds rows to a specified table.

Category: Data Manipulation

Restriction: This statement is not supported in FedSQL programs that run in CAS.

Requirement: Salesforce requires insertions to use column names.

Data source: SAS data set, SPD Engine data set, SPD Server table, Aster, DB2 under UNIX and PC, Google BigQuery, Greenplum, HAWQ, HDMD, Hive, Impala, JDBC, MDS, MongoDB, MySQL, Netezza, ODBC, Oracle, PostgreSQL, Salesforce, SAP HANA, SAP IQ, Spark, Teradata, Vertica, Yellowbrick

Notes: INSERT operations are not supported for MongoDB tables that contain structural ID values.

SAS 9.4, SAS Viya, and SAS Federation Server can support different data sources. For support information: SAS 9.4 and SAS Viya users, see "Data Source Support" in *Base SAS Procedures Guide*; SAS Federation Server users, see SAS Federation Server Documentation.

Syntax

Form 1: **INSERT** INTO *table*
{VALUES (*value* | NULL [, …*value* | NULL])};

Form 2: **INSERT** INTO *table*
{ (*column* [, …*column*]) VALUES (*value* | NULL [, …*value* | NULL]) };

Form 3: **INSERT** INTO *table*
{ (*column* [, …*column*]) [*query-expression*] };

Arguments

table

specifies the name of a table into which you are inserting rows.

Restriction If row-level permissions are in effect for the table, you cannot insert rows into the table. Row-level security is available only with SAS Federation Server. For more information about row-level security, see the SAS Federation Server documentation.

value

specifies a data value to insert into the table.

Restriction There must be one *value* for each column in the table or for each column in the column list (*column*).

Requirement If columns are specified in the column list (*column*), the data values list should be in the same order as the column list.

Tip You can use parameter arrays in the INSERT statement.

column

specifies the column into which you are inserting data.

Tip You can specify more than one column. The columns do not have to be in the same order as they appear in the table as long as the order of the column list and the data values list match one another.

<query-expression>

specifies any valid query expression that returns rows and where the number of columns in each row is the same as the number of items to be inserted.

See "SELECT Statement" on page 952

Details

Form 1: INSERT Statement without Column Names

This form of the INSERT statement that uses the VALUES clause without specifying column names can be used to insert lists of values into a table. You specify a value for each column in the table. One row is inserted for each VALUES clause. The order of the values in the VALUES clause should match the order of the columns in the table.

The following code fragment shows how the INSERT statement could be used with the VALUES clause to insert data values into the Customers table that was created in "CREATE TABLE Statement" on page 915. The Customers table has columns CustId, Name, Address, City, State, Country, PhoneNumber, and InitOrder.

```
insert into customers values (1,'Peter Frank', '300 Rock Lane', 'Boulder', 'CO',
'United States',These statements add data values to the Sales table that was created in the  '3
insert into customers values (2,'Jim Stewart', '1500 Lapis Lane', 'Little Rock',
'AR', 'United States', '8705553978', date '2012-03-20');
insert into customers values (3,'Janet Chien', '75 Jujitsu', 'Nagasaki', '', 'Japan',
```

```
'01181956879932', date '2012-06-07');
insert into customers values (4,'Qing Ziao', '10111 Karaje', 'Tokyo', '', 'Japan',
'0118136774351', date '2012-10-12');
insert into customers values (5,'Humberto Sertu', '876 Avenida Blanca', 'Buenos Aires',
'', 'Argentina','01154118435029', date '2012-12-15');
```

These statements add data values to the Sales table that was created in the "CREATE TABLE Statement" on page 915. The Sales table has columns ProdID, CustID, Totals, and Country.

```
insert into sales values (3234, 1, 189400, 'United States');
insert into sales values (1424, 3, 555789, 'Japan');
insert into sales values (3421, 4, 781183, 'Japan');
insert into sales values (3421, 2, 2789654, 'United States');
insert into sales values (3975, 5, 899453, 'Argentina');
```

Note that character values are enclosed in quotation marks.

Form 2: INSERT Statement with Column Names

This form of the INSERT statement that uses the VALUES clause with specific column names can also be used to insert lists of values into a table. You give values for the columns specified in the list of column names. The column names do not have to be in the same order as they appear in the table, but the order of the values in the VALUES clause must match the order of the column names in the INSERT column list. One row is inserted for each VALUES clause.

In the following code fragment, column names are used to add values to the Products table that was created in the "CREATE TABLE Statement" on page 915..

```
insert into products (prodid, product) values (1424, 'Rice');
insert into products (prodid, product) values (3421, 'Corn');
insert into products (prodid, product) values (3234, 'Wheat');
insert into products (prodid, product) values (3485, 'Oat');
```

The column list enables you to add a row that contains values for specified columns. If you do not specify the column list, values for every column in the table must be provided by the VALUES clause.

Form 3: INSERT Statement with a Query Expression

This form of the INSERT statement uses a query expression that inserts rows that were selected from another table or tables. The order of the values returned by the query expression should match the order of the columns that are specified in the list of column names. All the rows that are returned by the query expression are inserted into the table. The following restrictions should be noted when you use this form of the INSERT statement.

- If you do not specify the column list, values for every column in the table must be provided by the query expression.
- If you do not specify all columns in the column list, a null value is inserted for each column that is not listed.
- If the data type of the column that is being added does not match the data type of the column in the table, an automatic type conversion is attempted.

Here is an example of an INSERT statement that contains a query expression.

```
insert into coldtemps select * from worldtemps
 where AvgLow >=40;
```

This example queries an existing table WorldTemps and inserts the rows that are returned by the query into a new table, ColdTemps. The ColdTemps table has the same column names and data types as the WorldTemps table. The rows that are inserted into table ColdTemps include countries from the table WorldTemps that have an average low temperature less than or equal to 40 degrees. For information about table WorldTemps, see "WorldTemps" on page 1184.

Inserting Date, Time, and Timestamp Values

For information about inserting date, time, and timestamp values, see "Dates and Times in FedSQL" on page 58.

Comparisons

The DELETE statement enables you to delete rows from a table. The UPDATE statement enables you to change rows in a table. The INSERT statement enables you to insert new rows into a table.

See Also

Statements:

- "DELETE Statement" on page 936
- "SELECT Statement" on page 952
- "UPDATE Statement" on page 984

ROLLBACK Statement

Rolls back transaction changes to the beginning of the transaction.

Category:	Data Control
Restrictions:	This statement is not supported in FedSQL programs that run in CAS. The ROLLBACK statement has an effect only when autocommit functionality is off.
Data source:	Greenplum, HAWQ, MDS, Microsoft SQL Server, MySQL, ODBC, Oracle, SAP HANA, SAP IQ, Teradata, Vertica
Note:	SAS 9.4, SAS Viya, and SAS Federation Server can support different data sources. For support information: SAS 9.4 and SAS Viya users, see "Data Source Support" in *Base SAS Procedures Guide*; SAS Federation Server users, see SAS Federation Server Documentation.

Syntax

ROLLBACK [TRANSACTION];

Details

When your program has completed all of the statements in the transaction, you must explicitly terminate the transaction using COMMIT or ROLLBACK. You use a ROLLBACK statement to roll back, or undo, the changes that have been made since the start of the transaction.

You cannot roll back the changes to the database after a COMMIT statement is executed.

The ROLLBACK statement has an effect only when autocommit functionality is off. For most data sources, autocommit functionality is on by default. See the server administration documentation for information about how to turn off autocommit functionality. For example, see *SAS Federation Server: Administrator's Guide* for the appropriate connection option to the FedSQL driver. For the FEDSQL procedure, see *Base SAS Procedures Guide*.

Comparisons

The COMMIT statement takes all the changes that have been performed since the start of the transaction and makes them a permanent part of the database. The ROLLBACK statement causes all the changes that were made by the transaction to be rolled back to the start of the transaction.

See Also

Statements:

- "BEGIN Statement" on page 911
- "COMMIT Statement" on page 912

SELECT Statement

Retrieves columns and rows of data from tables.

Categories: Data Definition
Data Manipulation
Data Retrieval

Restriction: The full functionality of the SELECT statement is not supported in FedSQL programs that run in CAS. For CAS information, see "SELECT Statement" in *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*.

Data source: SAS data set, SPD Engine data set, SPD Server table, Amazon Redshift, Aster, DB2 under UNIX and PC, Google BigQuery, Greenplum, HAWQ, HDMD, Hive, Impala, JDBC, MDS, Microsoft SQL Server, MongoDB on Linux for x64, MySQL, Netezza, ODBC, Oracle, PostgreSQL, Salesforce, SAP, SAP HANA, SAP IQ, Spark, Teradata, Vertica, Yellowbrick

Notes: MongoDB tables are not relational tables and differ from relational tables in significant ways. In the August 2021 update of SAS/ACCESS on SAS 9.4M7 and SAS Viya 3.5, SAS/ACCESS software adds support for MongoDB structural IDs. See information about MongoDB in *SAS/ACCESS for Nonrelational Databases: Reference* before querying tables in MongoDB tables.

SAS 9.4, SAS Viya, and SAS Federation Server can support different data sources. For support information: SAS 9.4 and SAS Viya users, see "Data Source Support" in *Base SAS Procedures Guide*; SAS Federation Server users, see SAS Federation Server Documentation.

Syntax

The main clauses of the SELECT statement can be summarized as follows.

```
SELECT <select-list>
   FROM <table-specification>
   [WHERE <search-condition>]
   [GROUP BY <grouping-column>]
   [HAVING <search-condition>]
   [ORDER BY <sort-specification>]
   [LIMIT {count | ALL}]
   [OFFSET number]
;
```

The detailed syntax of the SELECT statement is as follows.

```
<query-expression>
   [ORDER BY <sort-specification> [, …<sort-specification>]];

<query-expression>::=
   {<query-specification> | <query-expression>}
      {UNION | EXCEPT | INTERSECT} {[ALL][CORRESPONDING [BY (column[,
        … column])]]} {<query-specification> | <query-expression>}

<query-specification>::=
   SELECT [ALL | DISTINCT] <select-list> <table-expression>

<select-list>::=
   *
   | column [AS column-alias]
   | expression [AS column-alias]
   | table.*
   | table-alias.*
```

```
    | table.column [AS column-alias]
<table-expression>::=
    FROM <table-specification> [, …<table-specification>]
    [WHERE <search-condition>]
    [GROUP BY <grouping-column> [, …<grouping-column>]]
    [HAVING <search-condition>]
<table-specification>::=
    table [[AS] alias]
    | CONNECTION TO catalog (<native-syntax>) [[AS] table-alias]
    | (<query-specification>) [AS] alias
    | <joined-table>
<joined-table>::=
    <cross-join>
    | <qualified-join>
    | <natural-join>
    <cross-join>::=
        <table-specification> CROSS JOIN <table-specification>
    <qualified-join>::=
        <table-specification> [<join-type>] JOIN <table-specification> <join-
          specification>
    <natural-join>::=
        <table-specification> NATURAL [<join-type>] JOIN <table-specification>
    <join-type>::=
        INNER
        | LEFT [OUTER]
        | RIGHT [OUTER]
        | FULL [OUTER]
    <join-specification>::=
        ON <search-condition>
        | USING (column [, …column])
<search-condition>::=
    {
        [NOT] {<sql-expression> | (<search-condition>)}
            [{AND | OR} [NOT] {<sql-expression> | (<search-condition>)}]
      }
        [,... {[NOT] {<sql-expression> | (<search-condition>)}
          [{AND | OR} [NOT] {<sql-expression> | (<search-condition>)}]}]
<sql-expression>::=
    expression {operator | predicate} expression
<sort-specification>::=
    {order-by-expression [ASC | DESC]} [, …order-by-expression [ASC | DESC]]
<grouping-column>::=
    column [, …column]
    | column-position-number
```

| <sql-expression>

Arguments

See the following sections for syntax argument descriptions.

- “SELECT Clause” on page 956
- “FROM Clause” on page 957
- “WHERE Clause” on page 967
- “GROUP BY Clause” on page 968
- “HAVING Clause” on page 970
- “ORDER BY Clause” on page 971
- “LIMIT Clause” on page 976
- “OFFSET Clause” on page 976
- “UNION Operator” on page 977
- “EXCEPT Operator” on page 979
- “INTERSECT Operator” on page 980

Details

Overview

The SELECT statement can be used in two ways.

- The single row SELECT statement, which can be executed by itself, returns only one row. For example:

  ```
  select 42;
  select 42 as x;
  ```

  The first code fragment returns a single column that contains the value 42. The column is named “column”. The second code fragment returns a similar column. However, the column is named “x”.

- A query specification begins with the SELECT keyword (called a SELECT clause) and cannot be used by itself. It reads column values from one or more tables and enables you to define conditions for the data that will be returned from the tables. It must be used as a part of another SQL statement and can return more than one row. A query specification creates a virtual table. For example:

  ```
  select column(s)
  from table(s)
  where condition(s);
  ```

The order of clauses in the SELECT statement is important. The optional clauses can be omitted but, when used, they must appear in the appropriate order. A SELECT statement can be specified within a SELECT statement (called a subquery). The ORDER BY, OFFSET, and LIMIT clauses can be used only on the outermost SELECT of a SELECT statement.

A view can be specified in the SELECT statement wherever a table can be specified. FedSQL supports native DBMS views and FedSQL views. It does not support PROC SQL views.

Note: There is no limit on the number of tables that you can reference in a FedSQL query. However, queries with a large number of table references can cause performance issues.

SELECT Clause

Description

Lists the columns that will appear in a virtual result set.

Syntax

SELECT [ALL | DISTINCT] <select-list>

<select-list>::=

*

| *column* [AS *column-alias*]

| <query-specification>

| <sql-expression> [AS *column-alias*]

| *table*.*

| *table-alias*.*

| *table*.*column* [AS *column-alias*]

Arguments

ALL
includes all rows, including duplicate rows in the result set.

DISTINCT
eliminates duplicate rows in the result set.

<select-list>
specifies the columns to be selected for the result set.

*
selects all columns in the table that is listed in the FROM clause.

column-alias
assigns a temporary, alternate name to the column.

***column* [AS *column-alias*]**
selects a single column. When [AS *column-alias*] is specified, assigns the column alias to the column.

<query-specification>
specifies an embedded SELECT subquery.

See "Overview of Subqueries" on page 51

<sql-expression> [AS *column-alias*]
derives a column name from an expression.

See "<sql-expression> Expression" on page 894

table*.
selects all columns in the table.

table-alias*.
selects all columns in the table.

See "Table Aliases" on page 959

***table.column* [AS *column-alias*]**
selects a single column from the specified table.

Asterisk (*) Notation

The asterisk (*) represents all columns of the table or tables that are listed in the FROM clause. When an asterisk is not prefixed with a table name, all the columns from all tables in the FROM clause are included; when it is prefixed (for example, *table*.* or *table-alias*.*), all the columns from only that table are included.

Column Aliases

A column alias is a temporary, alternate name for a column. Aliases are specified in the SELECT clause to name or rename columns in the result set in order to be clearer or easier to read. Aliases are often used to name a column that is the result of an arithmetic expression or summary function. An alias is one word only.

The keyword AS is required to distinguish a column alias from other column names.

Column aliases are optional, and each column name in the SELECT clause can have an alias. After you assign an alias to a column, you can use the alias to refer to that column in other clauses.

FROM Clause

Description

(Optional) specifies source tables.

Syntax

FROM <table-specification> [, …<table-specification>]

<table-specification>::=

table [AS *table-alias*]

| <CONNECTION TO *catalog*> (<native-syntax>) [AS *alias*]

| (<query-specification>) [AS *alias*]

| <joined-table>

<joined-table>::=

<cross-join>
| <qualified-join>
| <natural-join>
<cross-join>::=
<table-specification> CROSS JOIN <table-specification>
<qualified-join>::=
<table-specification> [<join-type>] JOIN <table-specification> <join-specification>
<natural-join>::=
<table-specification> NATURAL [<join-type>] JOIN <table-specification>
<join-type>::=
INNER
| LEFT [OUTER]
| RIGHT [OUTER]
| FULL [OUTER]
<join-specification>::=
ON <search-condition>
| USING (*column* [, …*column*])

Arguments

CONNECTION TO *catalog* (<native-syntax>) [[AS] *alias*]
specifies data from a DBMS catalog by using the SQL pass-through facility. You can use SQL syntax that the DBMS understands, even if that syntax is not valid in FedSQL. For more information, see "FedSQL Explicit Pass-Through Facility" on page 69.

CROSS JOIN
defines a join that is the Cartesian product of two tables.

See "Cross Joins" on page 962

JOIN
defines a join that enables you to filter the data by using a search condition or by using specific columns.

See "Qualified Joins" on page 962

NATURAL JOIN
defines a join that selects rows from two tables that have equal values in columns that share the same name and the same type.

See "Natural Joins" on page 967

(<query-specification>) [AS *alias*]
specifies an embedded SELECT subquery that functions as an in-line view. *alias* defines a temporary name for the in-line view and is required. An in-line view saves you a programming step. Rather than creating a view and referring to it in another query, you can specify the view in-line in the FROM clause.

See "Overview of Subqueries" on page 51

table
specifies the name of a table. The name can be in any of the forms described for "*table*" on page 917.

table-alias
specifies a temporary, alternate name for *table*. The AS keyword is optional.

INNER
specifies that only the subset of rows from the first table that matches rows from the second table are returned. Unmatched rows from both tables are discarded.

LEFT [OUTER]
specifies that matching rows and rows from the first table that do not match any row in the second table are returned.

RIGHT [OUTER]
specifies that matching rows and rows from the second table that do not match any row in the first table are returned.

FULL [OUTER]
species that all matching and unmatching rows from the first and second table are returned.

column
specifies the name of a column.

ON <search-condition>
specifies a condition join used to match rows from one table to another. If the search condition is satisfied, the matching rows are added to the result set.

See "<search-condition>" on page 982

USING (*column*[,...*column*])
specifies which columns to use in an inner or outer join.

See "ON and USING Clauses" on page 966

Overview

The FROM clause enables you to specify source tables. You can reference tables by specifying their table name, by submitting a query to a DBMS, by specifying an embedded SELECT subquery, or by specifying a join.

Table Aliases

A table alias is a temporary, alternate name for a table. Table aliases are used in joins to distinguish the columns of one table from those in the other table or tables and can make a query easier to read by abbreviating the table names. A table name or alias must be prefixed to a column name when you are joining tables that have matching column names. Column names in *reflexive joins* (joining a table with itself) must be prefixed with a table alias in order to distinguish which copy of the table the column comes from.

Table aliases are defined in the FROM clause. The AS keyword is used to distinguish the table alias from the table name. For example:

```
select b.* from x as b;
```

If you specify a table alias with a column name and do not define the table alias in the FROM clause, FedSQL returns an error message.

A table alias cannot be given an alias.

Joined Tables

When multiple table specifications are listed in the FROM clause, they are processed to form one table. The result set contains data from each contributing table. These queries are referred to as *joins*. Joins do not alter the original table.

Conceptually, when two tables are specified, each row of table A is matched with all the rows of table B to produce an internal or intermediate table. The number of rows in the intermediate table (*Cartesian*) is equal to the product of the number of rows in each of the source tables. The intermediate table becomes the input to the rest of the query in which some of its rows can be eliminated by the WHERE, ON, or USING clause or summarized by a function.

For an overview of FedSQL join operations, see “Join Operations” on page 30.

Specifying the Rows to Be Returned

The WHERE, ON, and USING clauses contain the conditions under which the rows in the Cartesian product are kept or eliminated in the result set. WHERE is used to select rows from inner joins. ON is used to select rows from inner or outer joins. USING is used to select specific columns to be included in the join. The condition is evaluated for each row from each table in the intermediate table described in “Joined Tables” on page 960. The row is considered to be a match if the result of the expression is true (a nonzero, nonmissing, or null value) for that row.

Simple Joins

The most basic type of join is simply a list of multiple tables, separated by commas, in the FROM clause of a SELECT statement. The following query joins the two tables GrainProducts and Sales that are shown in Appendix 4, “Tables Used in Examples,” on page 1169.

```
/* FedSQL code for simple join */
   title 'Simple Join - GrainProducts and Sales';
   select * from grainproducts, sales;
```

Output 7.1 *Simple Join - GrainProducts and Sales Table*

Simple Join - GrainProducts and Sales

PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
1424	Rice	3234	1	$189,400	United States
3421	Corn	3234	1	$189,400	United States
3234	Wheat	3234	1	$189,400	United States
3485	Oat	3234	1	$189,400	United States
1424	Rice	1424	3	$555,789	Japan
3421	Corn	1424	3	$555,789	Japan
3234	Wheat	1424	3	$555,789	Japan
3485	Oat	1424	3	$555,789	Japan
1424	Rice	3421	4	$781,183	Japan
3421	Corn	3421	4	$781,183	Japan
3234	Wheat	3421	4	$781,183	Japan
3485	Oat	3421	4	$781,183	Japan
1424	Rice	3421	2	$2,789,654	United States
3421	Corn	3421	2	$2,789,654	United States
3234	Wheat	3421	2	$2,789,654	United States
3485	Oat	3421	2	$2,789,654	United States
1424	Rice	3975	5	$899,453	Argentina
3421	Corn	3975	5	$899,453	Argentina
3234	Wheat	3975	5	$899,453	Argentina
3485	Oat	3975	5	$899,453	Argentina

Joining tables in this way returns the *Cartesian* product of the tables. Each row from the first table is combined with every row of the second table. The number of rows in the result set is equal to the number of rows in the first table multiplied by the number of rows in the second table.

The Cartesian product of a simple join can result in large, meaningless tables. You can subset a simple join by using a WHERE clause. This type of simple join is known as an *equijoin*. The following query subsets the previous table by matching the ID columns and creates the table shown in Output 7.61 on page 962.

```
/* FedSQL code for equijoin */
   title 'Equijoin - GrainProducts and Sales';
   select * from grainproducts, sales
      where grainproducts.prodid=sales.prodid;
```

In an equijoin, the comparison has to be an equality comparison. Multiple match criteria (not shown here) can be specified by using the AND operator. When multiple match criteria are specified, only rows meeting all of the equality tests are returned.

Output 7.2 *Equijoin - GrainProducts and Sales Table*

Equijoin - GrainProducts and Sales

PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
3234	Wheat	3234	1	$189,400	United States
1424	Rice	1424	3	$555,789	Japan
3421	Corn	3421	4	$781,183	Japan
3421	Corn	3421	2	$2,789,654	United States

Cross Joins

The cross join functions the same as a simple join; it returns the product of two tables. Like a Cartesian product, a cross join's output can be limited by a WHERE clause.

The following queries produce the same result.

```
select * from grainproducts, sales;
```

```
select * from grainproducts cross join sales;
```

Note: Do not use an ON clause with a cross join. An ON clause causes a cross join to fail. However, you can use a WHERE clause to subset the output.

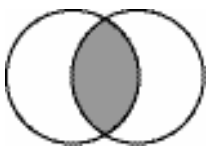
Qualified Joins

Qualified joins provide an easier way to control which rows appear in the result set. You can also further subset the result set with the ON or USING clause.

The two types of qualified joins are inner and outer.

Inner Joins

Figure 7.1 *Inner Join Diagram*



An *inner join* returns a result set for all the rows in one table that have one or more matching rows in another table. Using the GrainProducts and Sales tables, the following query matches the product ID columns of the two tables and creates the result set shown in Output 7.62 on page 963.

```
/* FedSQL code for inner join */
   title 'Inner Join - GrainProducts and Sales';
   select *
```

```
   from grainproducts inner join sales
      on grainproducts.prodid=sales.prodid;
```

Output 7.3 *Inner Join - GrainProducts and Sales Table*

Inner Join - GrainProducts and Sales

PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
3234	Wheat	3234	1	$189,400	United States
1424	Rice	1424	3	$555,789	Japan
3421	Corn	3421	4	$781,183	Japan
3421	Corn	3421	2	$2,789,654	United States

You can use the ON or USING clause instead of the WHERE clause to specify the column or columns on which you are joining the tables. However, you can continue to use the WHERE clause to subset the query result.

Note that an inner join with an ON or USING clause can provide the same functionality as listing tables in the FROM clause and specifying join columns with a WHERE clause (an equijoin). For example, these two sets of code use the inner join construction.

```
select *
   from grainproducts inner join sales
      on grainproducts.prodid=sales.prodid;
```

```
select *
   from grainproducts inner join sales
      using (prodid);
```

This code produces the same output as the previous code but uses the inner join construction.

```
select *
   from grainproducts, sales
      where grainproducts.prodid=sales.prodid;
```

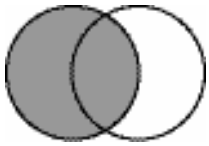
Outer Joins

Outer joins are inner joins that have been augmented with rows from one table that do not match with any row from the other table in the join. The result set includes rows that match and rows that do not match from the join's source tables. Nonmatching rows have null or missing values in the columns from the unmatched table. You can use the ON or USING clause instead of the WHERE clause to specify the column or columns on which you are joining the tables. However, you can continue to use the WHERE clause to subset the query result.

The three types of outer joins are left, right, and full.

Left Outer Joins

Figure 7.2 *Left Outer Join Diagram*



A left outer join lists matching rows and rows from the first table listed in the FROM clause that do not match any row in the second table listed in the FROM clause. Using the GrainProducts and Sales tables, the following code creates a table with matching rows from the GrainProducts and Sales tables and the unmatched rows from the GrainProducts table. Note that missing values are shown for Sales table data in the unmatched row from the GrainProducts table.

```
/* FedSQL code for left outer join */
title 'Left Outer Join - GrainProducts and Sales';
select *
   from grainproducts left outer join sales
      on grainproducts.prodid=sales.prodid;
```

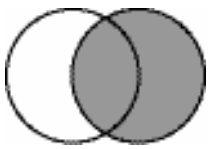
Output 7.4 *Left Outer Join - GrainProducts and Sales Table*

Left Outer Join - GrainProducts and Sales

PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
3234	Wheat	3234	1	$189,400	United States
1424	Rice	1424	3	$555,789	Japan
3421	Corn	3421	4	$781,183	Japan
3421	Corn	3421	2	$2,789,654	United States
3485	Oat	.	.	.	

Right Outer Joins

Figure 7.3 *Right Outer Join Diagram*



A right outer join lists matching rows and rows from the second table listed in the FROM clause that do not match any row in the first table listed in the FROM clause. Using the GrainProducts and Sales tables, the following code creates a table with matching rows from the GrainProducts and Sales tables and the unmatched rows from the Sales table. Note that missing values are shown for GrainProducts table data in the unmatched row from the Sales table.

```
/* FedSQL code for right outer join */
title 'Right Outer Join - GrainProducts and Sales';
select *
   from grainproducts right outer join sales
      on grainproducts.prodid=sales.prodid;
```

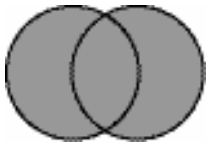
Output 7.5 *Right Outer Join - GrainProducts and Sales Table*

Right Outer Join - GrainProducts and Sales

PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
1424	Rice	1424	3	$555,789	Japan
3421	Corn	3421	4	$781,183	Japan
3421	Corn	3421	2	$2,789,654	United States
3234	Wheat	3234	1	$189,400	United States
.		3975	5	$899,453	Argentina

Full Outer Joins

Figure 7.4 *Full Outer Join Diagram*



A full outer join combines the left outer join and the right outer join. The result set contains both the matching and unmatching rows from the left and right tables. Using the GrainProducts and Sales tables, the following code creates a table with matching rows from the GrainProducts and Sales tables and the unmatched rows from the GrainProducts and Sales tables. Note that missing values are shown for data in the unmatched rows.

```
/* FedSQL code for full outer join */
title 'Full Outer Join - GrainProducts and Sales';
select *
   from grainproducts full outer join sales
      on grainproducts.prodid=sales.prodid;
```

Output 7.6 *Full Outer Join - GrainProducts and Sales Table*

Full Outer Join - GrainProducts and Sales

PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
3234	Wheat	3234	1	$189,400	United States
1424	Rice	1424	3	$555,789	Japan
3421	Corn	3421	4	$781,183	Japan
3421	Corn	3421	2	$2,789,654	United States
3485	Oat	.	.	.	
.		3975	5	$899,453	Argentina

ON and USING Clauses

You can use an ON clause with an expression that specifies a condition on which the join is based. The conditional expression can contain any predicate, although column names and comparison operators are most often used. The ON clause with an inner join is equivalent to a WHERE clause. The ON clause with an outer join (left, right, or full) is different from a WHERE clause. The ON clause with an outer join filters the rows and then includes the nonmatched rows with the null or missing values.

You can use a USING clause to specify one of two columns to include in the result set.

The difference between the ON clause and the USING clause is that you can use any conditional expression with the ON clause. The USING clause always implies an equality between the column names. For example, this ON clause eliminates **`United States`** from the results table.

```
/* FedSQL code for inner join with on clause */
title 'Inner Join - GrainProducts and Sales Outside US';
select *
   from grainproducts inner join sales
      on sales.country <> 'United States'
         AND grainproducts.prodid=sales.prodid;
```

Output 7.7 *Inner Join - GrainProducts and Sales outside the US*

Inner Join - GrainProducts and Sales Outside US

PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
1424	Rice	1424	3	$555,789	Japan
3421	Corn	3421	4	$781,183	Japan

Natural Joins

A *natural join* selects rows from two tables that have equal values in columns that share the same name and the same type. An error results if two columns have the same name but different types. You can perform an inner, left, right, or full natural join. If *join-type* is omitted when specifying a natural join, then INNER is implied. If like columns are not found, then a cross join is performed. You can use a WHERE clause to limit the output.

Using the GrainProducts and Sales tables, the following code performs a natural left outer join.

```
/* FedSQL code for natural left outer join */
title 'Natural Left Outer Join - GrainProducts and Sales';
select *
   from grainproducts natural left outer join sales;
```

Output 7.8 *Natural Left Outer Join - GrainProducts and Sales Table*

Natural Left Outer Join - GrainProducts and Sales

PRODID	PRODUCT	CUSTID	TOTALS	COUNTRY
3234	Wheat	1	$189,400	United States
1424	Rice	3	$555,789	Japan
3421	Corn	4	$781,183	Japan
3421	Corn	2	$2,789,654	United States
3485	Oat	.	.	

Notice that the *prodid* column appears only once in the result set.

Note: Do not use an ON clause with a natural join. An ON clause causes a natural join to fail. When using a natural join, an ON clause is implied, matching all like columns.

WHERE Clause

Description

Subsets the result set based on the specified search conditions.

Syntax

WHERE <search-condition>

Arguments

<search-condition>
specifies the conditions for the rows returned by the WHERE clause.

See "<search-condition>" on page 982

Details

The WHERE clause requires a search condition (one or more expressions separated by an operand or predicate) that specifies which rows are chosen for inclusion in the result set. When a condition is met (that is, the condition resolves to true), those rows are displayed in the result set. Otherwise, no rows are displayed.

Note: You cannot use aggregate functions that specify only one column. For example, you cannot use the following code.

```
where max(inventory1)>10000;
```

However, you can use this WHERE clause.

```
where max(inventory1, inventory2)>10000;
```

Note: If a column contains REAL or DOUBLE values, avoid using a WHERE clause with the = and the <> operators. REAL and DOUBLE values are approximate numeric data types and can give inaccurate results when used in a WHERE clause with the = and the <> operators. You should limit REAL and DOUBLE columns to comparisons with the > or < operator.

GROUP BY Clause

Description

Specifies how to group the data for summarizing.

Syntax

GROUP BY <grouping-column> [, …<grouping-column>]
<grouping-column>::=
column [, …*column*]
| *column-position-number*
| <sql-expression>

Arguments

column
specifies the name of a column or a column alias.

column-position-number
specifies a nonnegative integer that equates to a column position.

<sql-expression>
specifies a valid SQL expression.

See "<sql-expression> Expression" on page 894

Details

The GROUP BY clause groups data by a specified column or columns.

If the column or columns on which you are grouping contain missing or null values in some rows, SAS collects all the rows with missing or null values in the grouping columns into a single group.

You can specify more than one grouping column to get more detailed reports. If more than one grouping column is specified, then the first one determines the major grouping.

Integers can be substituted for column names in the GROUP BY clause. For example, if the grouping column is 2, then the results are grouped by values in the second column. Note that if you use a floating-point value (for example, 2.3) instead of an integer, then FedSQL ignores the decimal portion.

You can group the output by the values that are returned by an expression. For example, if X is a numeric variable, then the output of the following is grouped by the values of X.

```
select x, sum(y)
   from table1
   group by x;
```

Similarly, if Y is a character variable, then the output of the following is grouped by the values of Y.

```
select sum(x), y
   from table1
   group by y;
```

When you use a GROUP BY clause, you can also use an aggregate function in the SELECT clause or in a HAVING clause to instruct SAS in how to summarize the data for each group. When you use a GROUP BY clause without an aggregate function, SAS treats the GROUP BY clause as if it were an ORDER BY clause.

You can use the ORDER BY clause to specify the order in which rows are displayed in the result set. If you do not specify the ORDER BY clause, groups returned by the GROUP BY clause are not in any particular order.

Note: For an SPD Engine data set, utility files are used for certain operations that need extra space. The GROUP BY clause requires a utility file and the SPDEUTILLOC= system option allocates space for that utility file. For more information, see *SAS Scalable Performance Data Engine: Reference*.

Note: FedSQL GROUP BY syntax does not support remerging of summary statistics. The select list in a GROUP BY query can reference only grouping columns and aggregate functions. If you specify a column in the SELECT list that is

not specified in the GROUP BY clause, you get this error: `column name must be grouped or used in an aggregate function`.

> **TIP** In FedSQL, the results of an analysis can be integrated with other columns from the data by using a subquery. Specify the analysis in the subquery and then select the rest of the row based on the results of the subquery.

Here is an example of using a subquery to remerge data. The request uses a subquery to identify the oldest employee of each gender in a payroll table, and then it references the subquery in an outer query. Note that the subquery (t) is referenced in both the SELECT list and in the join condition.

```
select
p.IdNumber, p.Gender, p.Jobcode, p.Salary, p.Birth, p.Hired
   from
   (select Gender, min(Birth) as min_birth 1
     from sql.payroll
     group by Gender
     ) as t
   inner join sql.payroll as p on p.Gender=t.gender and
p.Birth=t.min_birth;
quit;
```

For more information, see "Rewriting a GROUP BY Remerge Query in FedSQL" on page 1207.

HAVING Clause

Description

Subsets grouped data based on specified search conditions.

Syntax

HAVING <search-condition>

Arguments

<search-condition>
specifies the conditions for the rows returned by the HAVING clause.

See "<search-condition>" on page 982

Details

The HAVING clause requires a search condition (one or more expressions separated by an operand or predicate) that specifies which rows are chosen for inclusion in the result set. A HAVING clause evaluates as either true or false for

each group in a query. You can use a HAVING clause with a GROUP BY clause to filter grouped data. The HAVING clause affects groups in a way that is similar to how a WHERE clause affects individual rows.

Queries that contain a HAVING clause usually also contain a GROUP BY clause, an aggregate function, or both. When you use a HAVING clause without a GROUP BY clause, SAS treats the HAVING clause as if it were a WHERE clause.

Table 7.1 *Differences between the HAVING Clause and WHERE Clause*

HAVING clause attributes	WHERE clause attributes
typically used to specify conditions for including or excluding groups of rows from a table	used to specify conditions for including or excluding individual rows from a table
must follow the GROUP BY clause in a query, if used with a GROUP BY clause	must precede the GROUP BY clause in a query, if used with a GROUP BY clause
affected by a GROUP BY clause; when there is no GROUP BY clause, the HAVING clause is treated like a WHERE clause	not affected by a GROUP BY clause
processed after the GROUP BY clause and any aggregate functions	processed before a GROUP BY clause, if there is one, and before any aggregate functions

ORDER BY Clause

Description

Specifies the order in which rows are returned in a result set.

Syntax

ORDER BY <sort-specification> [, …<sort-specification>];

<sort-specification>::=

{*order-by-expression* [ASC | DESC]} [, …*order-by-expression* [ASC | DESC]]

| {*order-by-expression* [COLLATE *collating-sequence-options*]}
[, …*order-by-expression* [COLLATE *collating-sequence-options*]]

Arguments

order-by-expression

specifies a column on which to sort. The sort column can be one of the following.

column

specifies the name of a column or a column alias.

column-position-number
specifies a nonnegative integer that equates to a column position.

<sql-expression>
specifies any valid SQL expression.

See "<sql-expression> Expression" on page 894

ASC
orders the data in ascending order. This is the default order; if ASC or DESC are not specified, the data is ordered in ascending order.

DESC
orders the data in descending order.

COLLATE *collating-sequence-options*
specifies linguistic collation, which sorts characters according to rules of the specified language. The rules and default collating sequence options are based on the language specified in the current locale setting. The implementation is provided by the International Components for Unicode (ICU) library and produces results that are largely compatible with the Unicode Collation Algorithms (UCA).

The *collating-sequence-options* argument can be one of the following values:

DANISH | FINNISH | ITALIAN | NORWEGIAN | POLISH | SPANISH | SWEDISH
sorts characters according to the language specified.

LINGUISTIC [*collating-rules*]
collating-rules can be one of the following values:

ALTERNATE_HANDLING=SHIFTED
controls the handling of variable characters like spaces, punctuation, and symbols. When this option is not specified (using the default value NON_IGNORABLE), differences among these variable characters are of the same importance as differences among letters. If the ALTERNATE_HANDLING option is specified, these variable characters are of minor importance.

Default NON_IGNORABLE

Tip The SHIFTED value is often used in combination with STRENGTH= set to Quaternary. In such a case, spaces, punctuation, and symbols are considered when comparing strings, but only if all other aspects of the strings (base letters, accents, and case) are identical.

CASE_FIRST=
specify order of uppercase and lowercase letters. This argument is valid for only TERTIARY, QUATERNARY, or IDENTICAL levels. The following table provides the values and information for the CASE_FIRST argument:

Table 7.2 *CASE_FIRST= Values*

Value	Description
UPPER	Sorts uppercase letters first, then the lowercase letters.
LOWER	Sorts lowercase letters first, then the uppercase letters.

COLLATION=
The following table lists the available COLLATION= values. If you do not select a collation value, then the user's locale-default collation is selected.

Table 7.3 COLLATION= Values

Value	Description
BIG5HAN	specifies Pinyin ordering for Latin and specifies big5 charset ordering for Chinese, Japanese, and Korean characters.
DIRECT	specifies a Hindi variant.
GB2312HAN	specifies Pinyin ordering for Latin and specifies gb2312han charset ordering for Chinese, Japanese, and Korean characters.
PHONEBOOK	specifies a telephone-book style for ordering of characters. Select PHONEBOOK only with the German language.
PINYIN	specifies an ordering for Chinese, Japanese, and Korean characters based on character-by-character transliteration into Pinyin. This ordering is typically used with simplified Chinese.
POSIX	is the Portable Operating System Interface. This option specifies a "C" locale ordering of characters.
STROKE	specifies a nonalphabetic writing style ordering of characters. Select STROKE with Chinese, Japanese, Korean, or Vietnamese languages. This ordering is typically used with Traditional Chinese.
TRADITIONAL	specifies a traditional style for ordering of characters. For example, select TRADITIONAL with the Spanish language.

LOCALE= *locale_name*
specifies the locale name in the form of a POSIX name(for example, ja_JP). For more information, see *SAS National Language Support (NLS): Reference Guide*

NUMERIC_COLLATION=
orders integer values within the text by the numeric value instead of characters used to represent the numbers.

Table 7.4 NUMERIC_COLLATION= Values

Value	Description
ON	Order numbers by the numeric value. For example, "8 Main St." would sort before "45 Main St.".

OFF	Order numbers by the character value. For example, "45 Main St." would sort before "8 Main St.".

Default OFF

STRENGTH=

The value of strength is related to the collation level. There are five collation-level values. The following table provides information about the five levels. The default value for strength is related to the locale.

Table 7.5 *STRENGTH= Values*

Value	Type of Collation	Description
PRIMARY or 1	PRIMARY specifies differences between base characters (for example, "a" < "b").	It is the strongest difference. For example, dictionaries are divided into different sections by base character.
SECONDARY or 2	Accents in the characters are considered secondary differences (for example, "as" < "às" < "at").	A secondary difference is ignored when there is a primary difference anywhere in the strings. Other differences between letters can also be considered secondary differences, depending on the language.
TERTIARY or 3	Upper and lowercase differences in characters are distinguished at the tertiary level (for example, "ao" < "Ao" < "aò").	A tertiary difference is ignored when there is a primary or secondary difference anywhere in the strings. Another example is the difference between large and small Kana.
QUATERNARY or 4	When punctuation is ignored at level 1–3, an additional level can be used to distinguish words with and without punctuation (for example, "ab" < "a-b" < "aB").	The quaternary level should be used if ignoring punctuation is required or when processing Japanese text. This difference is ignored when there is a primary, secondary, or tertiary difference.
IDENTICAL or 5	When all other levels are equal, the identical level is used as a tiebreaker. The Unicode code point values of the Normalization Form D (NFD) form of each string are compared at this level, just in	This level should be used sparingly, as only code point values differences between two strings is an extremely rare occurrence. For example, only Hebrew cantillation marks are distinguished at this level.

case there is no difference at levels 1–4.

Alias LEVEL=

Restriction Linguistic collation is not supported on platforms VMS on Itanium (VMI) or 64-bit Windows on Itanium (W64).

Tip The *collating-rules* must be enclosed in parentheses. More than one collating rule can be specified.

See "ICU Licence: ICU 1.8.1–ICU 57 and ICU4J 1.3.1–ICU4J 57" on page 1233.

The section on Linguistic Collation in *SAS National Language Support (NLS): Reference Guide*.

Refer to *http://www.unicode.org* for the Unicode Collation Algorithm (UCA) specification.

Details

The ORDER BY clause sorts the result of a query expression according to the order specified in that query. When this clause is used, the default ordering sequence is ascending, from the lowest value to the highest.

If an ORDER BY clause is omitted, then a particular order to the output rows, such as the order in which the rows are encountered in the queried table, cannot be guaranteed. Without an ORDER BY clause, the order of the output rows is determined by the internal processing of FedSQL, the default collating sequence of SAS, and your operating environment. Therefore, if you want your result set to appear in a particular order, then use the ORDER BY clause.

If more than one *order-by-expression* is specified (separated by commas), then the first one determines the major sort order.

Integers can be substituted for column names in the ORDER BY clause. For example, if the *order-by-expression* is 2, then the results are ordered by values in the second column. Note that if you use a floating-point value (for example, 2.3) instead of an integer, then FedSQL issues an error message.

In the ORDER BY clause, you can specify any column of a table that is specified in the FROM clause of a query expression, regardless of whether that column has been included in the query's SELECT clause. However, if SELECT DISTINCT is specified, or if the SELECT statement contains a UNION set operator, the sort column must appear in the query's SELECT clause.

Note: SAS missing values or null values are treated as the lowest possible values.

Note: For an SPD Engine data set, utility files are used for certain operations that need extra space. The ORDER BY clause requires a utility file and the Base SAS UTILLOC system option allocates space for that utility file. For more information, see *SAS Scalable Performance Data Engine: Reference*.

LIMIT Clause

Description

Specifies the number of rows that the SELECT statement returns.

Syntax

LIMIT {*count* | ALL}

Arguments

count
specifies the number of rows that the SELECT statement returns.

Tip *count* can be an integer or any simple expression that resolves to an integer value.

ALL
specifies that all rows are returned.

Details

The LIMIT clause can be used alone or in conjunction with the OFFSET clause. The OFFSET clause specifies the number of rows to skip before the SELECT statement starts to return rows.

Note: When you use the LIMIT clause, it is recommended that you use an ORDER BY clause to create an ordered sequence. Otherwise, you can get an unpredictable subset of a query's rows.

OFFSET Clause

Description

Specifies the number of rows to skip before the SELECT statement starts to return rows.

Syntax

OFFSET *number*

Arguments

number
specifies the number of rows to skip.

Tip *number* can be an integer or any simple expression that resolves to an integer value.

Details

The OFFSET clause can be used alone or in conjunction with the LIMIT clause. The OFFSET clause specifies the number of rows to skip before the SELECT statement starts to return rows.

Note: When you use the OFFSET clause, it is recommended that you use an ORDER BY clause to create an ordered sequence. Otherwise, you get an unpredictable subset of a query's rows.

UNION Operator

Description

Combines the result of two queries into a single result set.

Syntax

```
{<query-specification> | <query-expression>}
    UNION
        {
            [ALL] [CORRESPONDING [BY (column[, ...column]) ]]
          }
      {<query-specification> | <query-expression>}
```

Arguments

<query-specification> | <query-expression>
specifies one or more SELECT statements that produce a virtual result set.

See "SELECT Statement" on page 952

"Overview of Subqueries" on page 51

UNION
specifies that multiple result sets are combined and returned as a single result set.

ALL
(optional) specifies that all rows, including duplicates, are included in the result set.

CORRESPONDING
(optional) specifies to combine columns in the result set by column name or column list.

BY (column[,… column])
(optional) specifies that only these columns be included in the result set. Columns are identified by column name.

Requirement The column names must exist in both input tables.

Details

The UNION set operator produces a result set that contains all the unique rows that result from two queries. That is, the result set contains rows that are produced by the first query and the second query (and any optional subsequent queries that are separated by a UNION operator). The UNION set operator automatically eliminates duplicate rows from its result sets. The optional ALL keyword can preserve the duplicate rows, reduce the execution by one step, and thereby improve the query's performance. Use it when you want to display all the rows resulting from the query, rather than just the unique rows.

Columns are appended by position in the tables by default, regardless of the column names. However, the data type of the corresponding columns must match or be of a corresponding data type where length is the only differentiator, or the union does not occur. That is, CHAR can be merged with CHAR or VARCHAR, and INTEGER can be merged with BIGINT, INTEGER, SMALLINT, or TINYINT. However, CHAR or INTEGER cannot be merged with DOUBLE.

When columns of different lengths are merged, the length of the longer column is used. For example, if one column is CHAR(4) and the corresponding column is CHAR(10), then CHAR(10) is used.

The optional CORRESPONDING keyword causes FedSQL to match columns by column name instead of by position. The name matching is not case-sensitive. Formats are ignored unless they originate in the first query. Any columns that have non-matching column names are ignored. Use the CORRESPONDING keyword when the columns that you want to match have the same column name but exist in different positions in the input tables.

CORRESPONDING without a column list forms a result set that contains all columns that have the same name. The BY keyword restricts the columns that are returned to those whose names are specified in the column list. Columns that are not specified in the column list are dropped, even if the original queries specify SELECT *. Use the BY keyword to specify just one or more of the columns from both queries to be included in the result set.

The names of the columns in the result set are the names of the columns from the first query expression or query-specification unless a column (such as an expression) has no name in the first query expression or query-specification. Then, the name used depends on whether the SELECT statement result set is being returned to the client, or whether the SELECT statement is part of a CREATE TABLE AS statement. When the SELECT statement result set is being returned to the client, the name of an unnamed column in the result table is “column”. Unnamed columns in a CREATE TABLE AS statement are given generated names. The generated names are _expr_1, _expr_2, and so on.

See Also

“Query Expressions and Subqueries” on page 46

EXCEPT Operator

Description

Combines the results of two queries into a single result set that contains only rows that are unique to the first query.

Syntax

```
{<query-specification> | <query-expression>}
   EXCEPT
      {
         [ALL][CORRESPONDING [BY (column [, …column])]]
       }
{<query-specification> | <query-expression>}
```

Arguments

<query-specification> | <query-expression>
specifies one or more SELECT statements that produce a virtual result set.

See "SELECT Statement" on page 952

"Overview of Subqueries" on page 51

EXCEPT
specifies that multiple result sets are combined and only those rows that are in the first result set and not in the second result set are included.

ALL
(optional) specifies that all rows, including duplicates, are included in the result set.

CORRESPONDING
(optional) specifies to combine columns in the result set by column name or column list.

BY (*column*[, …*column*])
(optional) specifies that only these columns be included in the result set. Columns are identified by column name.

Requirement The column names must exist in both input tables.

Details

The EXCEPT set operator produces a result set that includes rows that exist in the results of the first query, except those that also exist in the results of the second query. Use the EXCEPT operator to identify results that are unique to the first query. If the intermediate result from the first query has at least one occurrence of a row that is not in the intermediate result of the second query, then that row (from the first query) is included in the result set.

The EXCEPT set operator automatically eliminates duplicate rows from its result sets. The optional ALL keyword can preserve the duplicate rows, reduce the execution by one step, and thereby improve the query's performance. Use it when you want to display all the rows resulting from the query, rather than just the unique rows.

Columns are evaluated based on their position in the input tables by default, regardless of the column names. However, the data type of the corresponding columns must match or be of a corresponding data type where length is the only differentiator; otherwise, the comparison does not occur. That is, CHAR can be evaluated against CHAR or VARCHAR, and INTEGER can be evaluated against BIGINT, INTEGER, SMALLINT, or TINYINT. However, CHAR or INTEGER cannot be evaluated against DOUBLE.

When columns of different lengths are merged, the length of the longer column is used.

The optional CORRESPONDING keyword causes FedSQL to match columns by column name instead of by position. The name matching is not case-sensitive. The column names display in the order that they are returned in the first query. Formats are ignored unless they originate in the first query. Any columns that have non-matching names are ignored. Use the CORRESPONDING keyword when the columns that you want to match have the same column name but exist in different positions in the input tables.

CORRESPONDING without a column list forms a result set that contains all columns that have the same name. The BY keyword restricts the columns that are returned to those whose names are specified in the column list. Columns that are not specified in the column list are dropped, even if the original queries specify SELECT *. Use the BY keyword to return only one or more columns from both queries in the result set.

The names of the columns in the result set are the names of the columns from the first query-expression or query-specification unless a column (such as an expression) has no name in the first query expression or query-specification. In such a case, the name of that column in the result set is “column”.

See Also

“Query Expressions and Subqueries” on page 46

INTERSECT Operator

Description

Combines the result of two queries into a single result set that contains only rows that are common to the queries.

Syntax

{<query-specification> | <query-expression>}
    INTERSECT

```
        {
            [ALL][CORRESPONDING [BY (column [, …column)]]]
          }
{<query-specification> | <query-expression>}
```

Arguments

<query-specification> | <query-expression>
specifies one or more SELECT statements that produces a virtual table.

See "SELECT Statement" on page 952

"Overview of Subqueries" on page 51

INTERSECT
specifies that multiple result sets are combined and only those rows that are common to the result sets are included.

ALL
(optional) specifies that all rows, including duplicates, are included in the result set.

CORRESPONDING
(optional) specifies to combine columns in the result set by column name or column list.

BY (*column*[, …*column*])
(optional) specifies that only these columns be included in the result set. Columns are identified by column name.

Requirement The column names must exist in both input tables.

Details

The INTERSECT operator produces a result set that has rows that are common to two queries.

The INTERSECT set operator automatically eliminates duplicate rows from its result sets. The optional ALL keyword is available to preserve the duplicate rows, reduce the execution by one step, and thereby improve the query's performance. Use it when you want to display all the rows resulting from the query, rather than just the unique rows.

Columns are evaluated based on their position in the input tables by default, regardless of the column names. However, the data type of the corresponding columns must match or be of a corresponding data type where length is the only differentiator, or the comparison does not occur. That is, CHAR can be evaluated against CHAR or VARCHAR, and INTEGER can be evaluated against BIGINT, INTEGER, SMALLINT, or TINYINT. However, CHAR or INTEGER cannot be evaluated against DOUBLE.

When columns of different lengths are merged, the length of the longer column is used.

The optional CORRESPONDING keyword causes FedSQL to match columns by column name instead of by position. The name matching is not case-sensitive. The column names display in the order that they are returned in the first query. Formats are ignored unless they originate in the first query. Any columns that have non-

matching names are ignored. Use the CORRESPONDING keyword when the columns that you want to match have the same column name but exist in different positions in the input tables.

CORRESPONDING without a column list forms a result set that contains all columns that have the same name. The BY keyword restricts the columns that are returned to those whose names are specified in the column list. Columns that are not specified in the column list are dropped, even if the original queries specify SELECT *. Use the BY keyword to return only one or more columns from both queries in the result set.

The names of the columns in the result set are the names of the columns from the first query-expression or query-specification unless a column (such as an expression) has no name in the first query expression or query-specification. In such a case, the name of that column in the result set is “column”.

See Also

“Query Expressions and Subqueries” on page 46

<search-condition>

Description

Is a combination of one or more operators and predicates that specifies which rows are chosen for inclusion in the result set.

Syntax

```
<search-condition>::=
    {
        [NOT] {<sql-expression> | (<search-condition>)}
            [{AND | OR} [NOT] {<sql-expression> | (<search-condition>)}]
      }
        [, …{[NOT] {<sql-expression> | (<search-condition>)}
          [{AND | OR} [NOT] {<sql-expression> | (<search-condition>)}]}]
<sql-expression>::=
    expression {operator | predicate} expression
```

Arguments

NOT

negates a Boolean condition. This table outlines the outcomes when you compare true and false values using the NOT operator.

Table 7.6 Truth Table for the NOT Operator

NOT	Result
True	False
False	True
Unknown	Unknown

AND

combines two conditions by finding observations that satisfy both conditions. This table outlines the outcomes when you compare TRUE and FALSE values using the AND operator.

Table 7.7 Truth Table for the AND Operator

AND	*True*	*False*	*Unknown*
True	True	False	Unknown
False	False	False	False
Unknown	Unknown	False	Unknown

OR

combines two conditions by finding observations that satisfy either condition or both. This table outlines the outcomes when you compare TRUE and FALSE values using the OR operator.

Table 7.8 Truth Table for the OR Operator

OR	*True*	*False*	*Unknown*
True	True	True	True
False	True	False	Unknown
Unknown	True	Unknown	Unknown

<sql-expression>

specifies any valid SQL expression.

See "<sql-expression> Expression" on page 894

Details

The search condition specifies which rows are returned in a result set for a SELECT statement. Within the SELECT statement, the search condition is used in the WHERE clause, the HAVING clause, and the ON clause with joins.

The order of precedence for the logical operators is NOT, AND, and then OR, but you can override the order by using parentheses. Everything within the parentheses is evaluated first to yield a single value before that value can be used by any operator outside of the parentheses.

There can be precision issues when REAL values are involved in a search condition. To avoid these issues, it is recommended that you define or create a cast for the value in the condition. In other words, wherever the value is used in the original query, force a cast with this syntax.

CAST (*value* AS *type)*

Here are some examples:

```
select * from test where x = cast (1.0e20 as real);
select cast (1.0e20 as real) from test;
select cast (col1 as real) from test;
```

Note: The search condition is also used with the "UPDATE Statement" on page 984 and the "DELETE Statement" on page 936. For the UPDATE statement, the search specification specifies which rows are updated. For the DELETE statement, the search specification specifies which rows are deleted.

See Also

Concepts:

- "FedSQL Explicit Pass-Through Facility" on page 69

Statements:

- "DELETE Statement" on page 936
- "UPDATE Statement" on page 984

UPDATE Statement

Modifies a column's values in existing rows of a table.

Category:	Data Manipulation
Restriction:	This statement is not supported in FedSQL programs that run in CAS.
Data source:	SAS data set, SPD Engine data set, SPD Server table, Amazon Redshift, Aster, DB2 under UNIX and PC, Google BigQuery, Greenplum, JDBC, MDS, Microsoft SQL Server, MongoDB, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, SAP IQ, Teradata, Vertica, Yellowbrick
Note:	SAS 9.4, SAS Viya, and SAS Federation Server can support different data sources. For support information: SAS 9.4 and SAS Viya users, see "Data Source Support" in *Base SAS Procedures Guide*; SAS Federation Server users, see SAS Federation Server Documentation.

Syntax

UPDATE *table*

{

SET *column*=<sql-expression> [, …*column*=<sql-expression>]

| *column*=*value-expression* [, …*column*=*value-expression*]

}

[WHERE <sql-expression> | *value-expression*];

Arguments

table

specifies a table name.

Restriction A Teradata table must have a primary key defined to be updated with the UPDATE statement.

column

specifies a column name.

<sql-expression>

specifies any valid SQL expression.

Note Only single variable operators are allowed in MongoDB subqueries. FedSQL does not support the use of subqueries as inner queries to delete data in MongoDB child tables or metadata tables.

See "<sql-expression> Expression" on page 894

value-expression

specifies any valid value expression.

Tip You can use parameter arrays in the INSERT statement.

Details

The UPDATE statement changes the values in all or part of an existing row in a table.

The SET clause specifies which columns to modify and the values to assign them. Columns that are not SET retain their previous values. In the SET clause, a column reference on the left side of the equal sign can also appear as part of the expression on the right side of the equal sign. The SET clause uses the current value of the column in the expression. For example, you could use this expression to give employees a $1,000 holiday bonus.

```
update payroll set salary = salary + 1000;
```

The WHERE clause enables you to choose which rows to update. If you omit the WHERE clause, then all the rows are updated. When you use a WHERE clause, only the rows that meet the WHERE clause condition are updated.

Note: If row-level permissions are in effect for the table, you cannot update rows in the table. Row-level security is available only with SAS Federation Server. For more information about row-level security, see the SAS Federation Server documentation.

Comparisons

The DELETE statement enables you to delete rows from a table. The INSERT statement enables you to insert new rows into a table. The UPDATE statement enables you to change rows in a table.

See Also

Statements:

- "DELETE Statement" on page 936
- "INSERT Statement" on page 948
- WHERE clause in the "SELECT Statement" on page 952

8

FedSQL Statement Table Options

Overview of Statement Table Options

About FedSQL Statement Table Options

FedSQL statement table options specify actions that apply only to the table with which they appear. They enable you to perform operations such as specifying a password for a SAS file and renaming columns.

You can specify table options in a FedSQL statement in which you specify a table name, such as the CREATE TABLE, ALTER TABLE, or SELECT statement.

Restrictions

The availability and behavior of FedSQL statement options are data-source specific.

How Table Options Interact with Other Types of Options

Many types of options share the same name and have the same function. For example, you can compress a SAS data set by specifying the COMPRESS= table option as a connection string option and as a FedSQL statement table option.

When more than one type of option with the same function is specified, the software follows the following order of precedence:

1. FedSQL statement table option
2. data source connection string option

For example, if a connection string includes COMPRESS=NO, and a FedSQL statement includes the table option COMPRESS=YES, then the table option overrides the connection string specification for the named table.

If you are submitting FedSQL statements in a Base SAS session, the following order of precedence applies:

1. FedSQL statement option
2. LIBNAME statement option
3. data source connection string option
4. system option

That is, FedSQL statement table options override a LIBNAME statement option, which overrides a data source connection string option.

FedSQL Statement Table Option Syntax

Specify a FedSQL statement table option immediately after the table name, within braces (that is, { }) and including the keyword OPTIONS. To specify several table options, separate them with spaces or commas.

CAUTION
While specifying the syntax for table options, you cannot have a space between the left brace { and the OPTIONS keyword. A space results in a syntax error. For example, this statement is correct:

```
create table temp {options dbcreate_table_opts='primary index(b)'}
(a int, b int);
```

The following statement, however, is incorrect and results in an error message:

```
create table temp { options dbcreate_table_opts='primary index(b)'}
(a int, b int);
```

{OPTIONS *option-1=value* [… *option-n=value*]}

These examples show table options in FedSQL statements:

```
create table salary {options encrypt=yes read=green};

select * from salary {options read=green};
```

Understanding BULKLOAD Table Options

Table options that are related to bulk loading are available only when the FedSQL language processes the call, as opposed to when the driver uses the native SQL language. The FedSQL language processor is used over native SQL to process requests that create tables or insert data from multiple data sources. In order for the table options to be enforced, the data source that supports them must be specified in the CREATE TABLE or INSERT INTO part of the query. The FedSQL language processor is also used to process CREATE TABLE and INSERT statements for a single data source when the requests specify a SAS function.

Bulk loading is initiated with the BULKLOAD= table option. Data source-specific bulk-loading table options are specified in the BULKOPTS= table option.

FedSQL Statement Table Options by Data Source

The following table lists the table options that are supported in FedSQL programs by the data source that supports them. The options are listed alphabetically for each data source.

Table 8.1 *List of Supported Table Options by Data Source*

Data Source	Language Element	Category	Description
Amazon Redshift	"DBCREATE_TABLE_OPTS= Table Option" (p. 1035)	Table control	Allows database-specific options to be added to the CREATE TABLE statement.
Aster	"PARTITION_KEY= Table Option" (p. 1060)	Table control	Specifies the column name to use as the partition key for creating fact tables.
DB2 under UNIX and PC	"BL_ALLOW_READ_ACCESS= Table Option" (p. 1004)	Bulk loading	Specifies that the original table data is still visible to readers during bulk load.
	"BL_COPY_LOCATION= Table Option" (p. 1005)	Bulk loading	Specifies the directory to which DB2 saves a copy of the loaded data.

Data Source	Language Element	Category	Description
	"BL_CPU_PARALLELISM= Table Option" (p. 1006)	Bulk loading	Specifies the number of processes or threads to use when building table objects.
	"BL_DATA_BUFFER_SIZE= Table Option" (p. 1007)	Bulk loading	Specifies the total amount of memory to allocate for the bulk load utility to use as a buffer for transferring data.
	"BL_DISK_PARALLELISM= Table Option" (p. 1009)	Bulk loading	Specifies the number of processes or threads to use when writing data to disk.
	"BL_EXCEPTION= Table Option" (p. 1011)	Bulk loading	Specifies the exception table into which rows in error are copied.
	"BL_INDEXING_MODE= Table Option" (p. 1012)	Bulk loading	Specifies which scheme the DB2 load utility should use for index maintenance.
	"BL_LOAD_REPLACE= Table Option" (p. 1013)	Bulk loading	Specifies whether DB2 appends or replaces rows during bulk loading.
	"BL_LOG= Table Option" (p. 1014)	Bulk loading	Identifies a log file that contains information such as statistics and error information for a bulk load.
	"BL_OPTIONS= Table Option" (p. 1016)	Bulk loading	Passes options to the DBMS bulk-load facility, affecting how it loads and processes data.
	"BL_PORT_MAX= Table Option" (p. 1017)	Bulk loading	Sets the highest available port number for concurrent uploads.
	"BL_PORT_MIN= Table Option" (p. 1018)	Bulk loading	Sets the lowest available port number for concurrent uploads.
	"BL_RECOVERABLE= Table Option" (p. 1019)	Bulk loading	Specifies whether the LOAD process is recoverable.
	"BL_REMOTE_FILE= Table Option" (p. 1020)	Bulk loading	Specifies the base filename and location of DB2 LOAD temporary files.
	"BL_WARNING_COUNT= Table Option" (p. 1023)	Bulk loading	Specifies the maximum number of row warnings to allow before the load fails.

Data Source	Language Element	Category	Description
	"BULKLOAD= Table Option" (p. 1026)	Bulk loading	Determines whether SAS uses a DBMS facility to insert data into a DBMS table.
	"BULKOPTS= Table Option" (p. 1028)	Bulk loading	Container for bulk load options. This option must follow BULKLOAD=YES.
	"DBCREATE_TABLE_OPTS= Table Option" (p. 1035)	Table control	Allows database-specific options to be added to the CREATE TABLE statement.
Google BigQuery	"BULKLOAD= Table Option" (p. 1026)	Bulk loading	Determines whether SAS uses a DBMS facility to insert data into a DBMS table.
	"BULKOPTS= Table Option" (p. 1028)	Bulk loading	Container for bulk load options. This option must follow BULKLOAD=YES.
	"DBCREATE_TABLE_OPTS= Table Option" (p. 1035)	Table control	Allows database-specific options to be added to the CREATE TABLE statement.
	"KMS_PATH= Table Option" (p. 1053)	Table access	Specifies the location of a key file resource within a project.
	"LOCATION= Table Option" (p. 1054)	Table access	Specifies the location of the data set.
Greenplum	"DBCREATE_TABLE_OPTS= Table Option" (p. 1035)	Table control	Allows database-specific options to be added to the CREATE TABLE statement.
	"GP_DISTRIBUTED_BY= Table Option" (p. 1047)	Table control	Specifies the distribution key for the table being created.
HAWQ	"DBCREATE_TABLE_OPTS= Table Option" (p. 1035)	Table control	Allows database-specific options to be added to the CREATE TABLE statement.
	"GP_DISTRIBUTED_BY= Table Option" (p. 1047)	Table control	Specifies the distribution key for the table being created.
Hive	"DBCREATE_TABLE_OPTS= Table Option" (p. 1035)	Table control	Allows database-specific options to be placed after the CREATE TABLE statement.

Data Source	Language Element	Category	Description
	"POST_TABLE_OPTS= Table Option" (p. 1064)	Table control	Allows database-specific options to be placed after the table name in a CREATE TABLE statement.
	"PRE_TABLE_OPTS= Table Option" (p. 1065)	Table control	Allows database-specific options to be placed before the table name in a CREATE TABLE statement.
Impala	"BULKLOAD= Table Option" (p. 1026)	Bulk loading	Determines whether SAS uses a DBMS facility to insert data into a DBMS table.
	"BULKOPTS= Table Option" (p. 1028)	Bulk loading	Provides a container for bulk load options.
	"CONFIG= Table Option" (p. 1032)	Bulk loading	Specifies a file or path name for Hadoop configuration path resolution.
	"DATAFILE= Table Option" (p. 1033)	Bulk loading	Specifies an alternate name and location for the temporary HDFS file.
	"PASSWORD= Table Option" (p. 1061)	Bulk loading	Specifies the password for the HDFS user.
	"PICKLIST= Table Option" (p. 1061)	Bulk loading	Specifies the picklist to use for the bulk-loading operation.
	"USER= Table Option" (p. 1098)	Bulk loading	Specifies the HDFS user name.
MDS	"BULKLOAD= Table Option" (p. 1026)	Bulk loading	Determines whether SAS uses a DBMS facility to insert data into a DBMS table.
Microsoft SQL Server	"DBCREATE_INDEX_OPTS= Table Option" (p. 1034)	Table control	Allows database-specific options to be added to the CREATE INDEX statement.
	"DBCREATE_TABLE_OPTS= Table Option" (p. 1035)	Table control	Allows database-specific options to be added to the CREATE TABLE statement.
MySQL	"DBCREATE_TABLE_OPTS= Table Option" (p. 1035)	Table control	Allows database-specific options to be added to the CREATE TABLE statement.
Netezza	"DBCREATE_TABLE_OPTS= Table Option" (p. 1035)	Table control	Allows database-specific options to be added to the CREATE TABLE statement.

Data Source	Language Element	Category	Description
ODBC	"DBCREATE_INDEX_OPTS= Table Option" (p. 1034)	Index control	Allows database-specific options to be added to the CREATE INDEX statement.
	"DBCREATE_TABLE_OPTS= Table Option" (p. 1035)	Table control	Allows database-specific options to be added to the CREATE TABLE statement.
Oracle	"BL_DEFAULT_DIR= Table Option" (p. 1008)	Bulk loading	Specifies where bulk load creates all intermediate files.
	"BL_ERRORS= Table Option" (p. 1010)	Bulk loading	Specifies that after the indicated number of errors is received, that the load should stop.
	"BL_LOAD= Table Option" (p. 1013)	Bulk loading	Specifies that after the indicated number of rows is loaded, that the load should stop.
	"BL_LOGFILE= Table Option" (p. 1015)	Bulk loading	Specifies the filename for the bulk load log file.
	"BL_PARALLEL Table Option" (p. 1017)	Bulk loading	Specifies whether to perform a parallel bulk load.
	"BL_RECOVERABLE= Table Option" (p. 1019)	Bulk loading	Determines whether the LOAD process is recoverable.
	"BL_SKIP= Table Option" (p. 1021)	Bulk loading	Specifies to skip the indicated number of rows before starting the bulk load.
	"BL_SKIP_INDEX_MAINTENANCE= Table Option" (p. 1021)	Bulk loading	Specifies whether to perform index maintenance on the bulk load.
	"BL_SKIP_UNUSABLE_INDEXES= Table Option" (p. 1022)	Bulk loading	Specifies whether to skip index entries that are in an unusable state and continue with the bulk load.
	"BULKLOAD= Table Option" (p. 1026)	Bulk loading	Determines whether SAS uses a DBMS facility to insert data into a DBMS table.
	"BULKOPTS= Table Option" (p. 1028)	Bulk loading	Container for bulk load options. This option must follow BULKLOAD=YES.
	"DBCREATE_TABLE_OPTS= Table Option" (p. 1035)	Table control	Allows database-specific options to be added to the CREATE TABLE statement.

Data Source	Language Element	Category	Description
	"ORHINTS= Table Option" (p. 1056)	Data control	Specifies Oracle hints to pass to Oracle from FedSQL.
	"ORNUMERIC= Table Option" (p. 1057)	Table control	Specifies how numbers read from or inserted into the Oracle NUMBER column are treated.
SAP HANA	"DBCREATE_TABLE_OPTS= Table Option" (p. 1035)	Table control	Allows database-specific options to be added to the CREATE TABLE statement.
	"TABLE_TYPE= Table Option" (p. 1070)	Table access	Specifies the type of table storage FedSQL will use when creating tables in SAP HANA.
SAS data set	"ALTER= Table Option" (p. 1002)	Table control	Assigns an ALTER password to a SAS data set that prevents users from replacing or deleting the file, and enables access to a read- or write-protected file.
	"BUFNO= Table Option" (p. 1024)	Table control	Specifies the number of buffers to be allocated for processing a SAS data set.
	"BUFSIZE= Table Option" (p. 1025)	Table control	Specifies the size of a permanent buffer page for an output SAS data set.
	"COMPRESS= Table Option" (p. 1030)	Table control	Specifies how rows are compressed in a new output data set.
	"ENCODING= Table Option" (p. 1038)	Table control	Specifies the encoding to use when creating a SAS data set.
	"ENCRYPT= Table Option" (p. 1039)	Table control	Specifies whether to encrypt an output SAS data set.
	"ENCRYPTKEY= Table Option" (p. 1042)	Table control	Specifies a key value for AES encryption.
	"EXTENDOBSCOUNTER= Table Option" (p. 1046)	Table control	Specifies whether to extend the maximum observation count in a new output SAS data file.
	"IDXNAME= Table Option" (p. 1048)	User control of index usage	Directs SAS to use a specific index to match the conditions of a WHERE clause.

Data Source	Language Element	Category	Description
	"IDXWHERE= Table Option" (p. 1050)	User control of index usage	Specifies whether SAS uses an index search or a sequential search to match the conditions of a WHERE clause.
	"LABEL= Table Option" (p. 1053)	Observation control	Specifies a label for a SAS data set.
	"LOCKTABLE= Table Option" (p. 1056)	Table control	Places shared or exclusive locks on tables.
	"POINTOBS= Table Option" (p. 1062)	Table control	Specifies whether SAS creates compressed data sets whose observations can be randomly accessed or sequentially accessed.
	"PW= Table Option" (p. 1066)	Table control	Assigns a READ, WRITE, and ALTER password to a SAS file, and enables access to a password-protected file.
	"READ= Table Option Table Option" (p. 1067)	Table control	Assigns a READ password to a SAS file that prevents users from reading the file, unless they enter the password.
	"REUSE= Table Option" (p. 1068)	Table control	Specifies whether new rows can be written to freed space in a compressed SAS data set.
	"TYPE= Table Option" (p. 1097)	Table control	Specifies the data set type for a specially structured SAS data set.
	"WRITE= Table Option Table Option" (p. 1100)	Table control	Assigns a WRITE password to a SAS file that prevents users from writing to the file or that enables access to a write-protected file.
SPD Engine data set	"ALTER= Table Option" (p. 1002)	Table control	Assigns an ALTER password to an SPD Engine data set that prevents users from replacing or deleting the file, and enables access to a read- or write-protected file.
	"ASYNCINDEX= Table Option" (p. 1003)	User control of index usage	Specifies to create indexes in parallel when creating multiple indexes on an SPD Engine data set.
	"COMPRESS= Table Option" (p. 1030)	Table control	Specifies to compress SPD Engine data sets on disk as they are being created.

Data Source	Language Element	Category	Description
	"ENCRYPT= Table Option" (p. 1039)	Table control	Specifies whether to encrypt an output SPD Engine data set.
	"ENCRYPTKEY= Table Option" (p. 1042)	Table control	Specifies a key value for AES encryption.
	"ENDOBS= Table Option" (p. 1045)	Observation control	Specifies the end observation number in a user-defined range of observations to be processed.
	"IDXWHERE= Table Option" (p. 1050)	User control of index usage	Specifies to use indexes when processing WHERE expressions in the SPD Engine.
	"IOBLOCKSIZE= Table Option" (p. 1051)	Table control	Specifies the size in bytes of a block of observations to be used in an I/O operation.
	"LABEL= Table Option" (p. 1053)	Observation control	Specifies a label for an SPD Engine data set.
	"PADCOMPRESS= Table Option" (p. 1058)	Table control	Specifies the number of bytes to add to compressed blocks in a data set opened for OUTPUT or UPDATE.
	"PARTSIZE= Table Option" (p. 1059)	Table control	Specifies the size of the data component partitions in an SPD Engine data set.
	"PW= Table Option" (p. 1066)	Table control	Assigns a READ, WRITE, and ALTER password to a SAS file, and enables access to a password-protected file.
	"READ= Table Option Table Option" (p. 1067)	Table control	Assigns a READ password to a SAS file that prevents users from reading the file, unless they enter the password.
	"STARTOBS= Table Option" (p. 1069)	Observation control	Specifies the starting observation number in a user-defined range of observations to be processed.
	"THREADNUM= Table Option" (p. 1096)	Table control	Specifies the maximum number of I/O threads the SPD Engine can spawn for processing an SPD Engine data set.

Data Source	Language Element	Category	Description
	"TYPE= Table Option" (p. 1097)	Table control	Specifies the data set type for a specially structured SPD Engine data set.
	"UNIQUESAVE= Table Option" (p. 1097)	User control of index usage	Specifies to save observations with nonunique key values (the rejected observations) to a separate data set when appending or inserting observations to data sets with unique indexes.
	"WHERENOINDEX= Table Option" (p. 1099)	User control of index usage	Specifies a list of indexes to exclude when making WHERE expression evaluations.
	"WRITE= Table Option Table Option" (p. 1100)	Table control	Assigns a WRITE password to a SAS file that prevents users from writing to the file or that enables access to a write-protected file.
SPD Server table	"COMPRESS= Table Option" (p. 1030)	Table control	Specifies to compress SPD Server tables on disk as they are being created.
	"ENCRYPT= Table Option" (p. 1039)	Table access	Specifies whether to encrypt an output SPD Server table.
	"ENCRYPTKEY= Table Option" (p. 1042)	Table access	Specifies a key for AES encryption.
	"ENDOBS= Table Option" (p. 1045)	Observation control	Specifies the end observation number in a user-defined range of observations to be processed.
	"IOBLOCKSIZE= Table Option" (p. 1051)	Table control	Specifies the size in bytes of a block of rows to be used in an I/O operation.
	"PARTSIZE= Table Option" (p. 1059)	Table control	Specifies the size of the data component partitions in an SPD Server table.
	"PW= Table Option" (p. 1066)	Table access	Enables you to access an SPD Server table that is protected by SAS Proprietary encryption.
	"STARTOBS= Table Option" (p. 1069)	Observation control	Specifies the starting observation number in a user-defined range of observations to be processed.

Data Source	Language Element	Category	Description
Spark	"DBCREATE_TABLE_OPTS= Table Option" (p. 1035)	Table control	Allows database-specific options to be placed after the CREATE TABLE statement.
	"POST_TABLE_OPTS= Table Option" (p. 1064)	Table control	Allows database-specific options to be placed after the table name in a CREATE TABLE statement.
	"PRE_TABLE_OPTS= Table Option" (p. 1065)	Table control	Allows database-specific options to be placed before the table name in a CREATE TABLE statement.
Teradata	"BULKLOAD= Table Option" (p. 1026)	Bulk loading	Determines whether SAS uses a DBMS facility to insert data into a DBMS table.
	"BULKOPTS= Table Option" (p. 1028)	Bulk loading	Container for bulk load options. This option must follow BULKLOAD=YES.
	"DBCREATE_TABLE_OPTS= Table Option" (p. 1035)	Table control	Allows database-specific options to be added to the CREATE TABLE statement.
	"TD_BUFFER_MODE= Table Option" (p. 1071)	Bulk loading	Specifies whether the LOAD method is used.
	"TD_CHECKPOINT= Table Option" (p. 1072)	Bulk loading	Specifies when the TPT operation issues a checkpoint or savepoint to the database.
	"TD_DATA_ENCRYPTION= Table Option" (p. 1072)	Bulk loading	Activates data encryption.
	"TD_DROP_ERROR_TABLE = Table Option" (p. 1073)	Bulk loading	Drops the log table at the end of the job, whether the job completed successfully or not.
	"TD_DROP_LOG_TABLE= Table Option" (p. 1074)	Bulk loading	Drops the error tables at the end of the job, whether the job completed successfully or not.
	"TD_DROP_WORK_TABLE= Table Option" (p. 1074)	Bulk loading	Drops the work table at the end of the job, whether the job completed successfully or not.
	"TD_ERROR_LIMIT= Table Option" (p. 1075)	Bulk loading	Specifies the maximum number of records that can be stored in an error table.

Data Source	Language Element	Category	Description
	"TD_ERROR_TABLE_1= Table Option" (p. 1076)	Bulk loading	Specifies a name for the first error table.
	"TD_ERROR_TABLE_2= Table Option" (p. 1077)	Bulk loading	Specifies a name for the second error table.
	"TD_LOG_MECH_TYPE= Table Option" (p. 1078)	Bulk loading	Specifies the logon mechanism for a bulk load.
	"TD_LOG_MECH_DATA= Table Option" (p. 1079)	Bulk loading	Specifies additional data for the logon mechanism.
	"TD_LOG_TABLE= Table Option" (p. 1080)	Bulk loading	Specifies the name of the restart log table.
	"TD_LOGDB= Table Option" (p. 1081)	Bulk loading	Specifies the database where the TPT utility tables are created.
	"TD_MAX_SESSIONS= Table Option" (p. 1081)	Bulk loading	Specifies the maximum number of logon sessions that TPT can acquire for a job.
	"TD_MIN_SESSIONS= Table Option" (p. 1082)	Bulk loading	Specifies the minimum number of sessions for TPT to acquire before a job starts.
	"TD_NOTIFY_LEVEL= Table Option" (p. 1083)	Bulk loading	Specifies the level at which log events are recorded.
	"TD_NOTIFY_METHOD= Table Option" (p. 1084)	Bulk loading	Specifies the method for reporting events.
	"TD_NOTIFY_STRING= Table Option" (p. 1085)	Bulk loading	Defines a string that precedes all messages sent to the system log.
	"TD_PACK= Table Option" (p. 1086)	Bulk loading	Specifies the number of statements to pack into a multistatement request.
	"TD_PACK_MAXIMUM= Table Option" (p. 1086)	Bulk loading	Enables the Stream operator to determine the maximum possible pack factor for the current Stream job.
	"TD_PAUSE_ACQ= Table Option" (p. 1087)	Bulk loading	Forces a pause between the acquisition phase and the application phase of a load job.

Data Source	Language Element	Category	Description
	"TD_SESSION_QUERY_BAND= Table Option" (p. 1088)	Bulk loading	Passes a string of user-specified name=value pairs for use by the TPT session.
	"TD_TENACITY_HOURS= Table Option" (p. 1088)	Bulk loading	Specifies the amount of time the TPT operator continues trying to log on to the Teradata database.
	"TD_TENACITY_SLEEP= Table Option" (p. 1089)	Bulk loading	Specifies the amount of time the TPT operator pauses, before retrying to log on to the Teradata database.
	"TD_TPT_OPER= Table Option" (p. 1090)	Bulk loading	Specifies the load operator used by the Teradata Parallel Transporter.
	"TD_TRACE_LEVEL=, TD_TRACE_LEVEL_INF= Table Options" (p. 1091)	Bulk loading	Specify the trace levels for driver tracing. TD_TRACE_LEVEL sets the primary trace level. TD_TRACE_LEVEL_INF sets the secondary trace level.
	"TD_TRACE_OUTPUT= Table Option" (p. 1092)	Bulk loading	Specifies the name of the external file used for trace messages.
	"TD_UNICODE_PASSTHRU= Table Option" on page 1093	Bulk loading	Allows pass-through characters (PTCs) to be imported into and exported from Teradata.
	"TD_WORKING_DB= Table Option" (p. 1095)	Bulk loading	Specifies the database where the table is to be created.
	"TD_WORK_TABLE= Table Option" (p. 1094)	Bulk loading	Specifies a name for the TPT work table.

Dictionary

ALTER= Table Option

Assigns an ALTER password to a data set that prevents users from replacing or deleting the file, and enables access to a Read- or Write-protected file.

Category: Table Control

Restriction: This table option is not supported on the CAS server.

Data source: SAS data set, SPD Engine data set

Note: Check your log after this operation to ensure that the password values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in *SAS Programmer’s Guide: Essentials*.

Syntax

ALTER= *alter-password*

Arguments

alter-password
specifies a password.

Restriction *alter-password* must be a valid SAS name.

Details

The ALTER= table option applies only to a SAS data set or an SPD Engine data set. You can use this option to assign a password or to access a Read-protected, Write-protected, or alter-protected file. When you replace a data set that is protected with an ALTER password, the new data set inherits the ALTER password.

Note: A SAS password does not control access to a SAS file beyond the SAS system. You should use the operating system-supplied utilities and file-system security controls to control access to SAS files outside of SAS.

ASYNCINDEX= Table Option

Specifies to create indexes in parallel when creating multiple indexes on an SPD Engine data set.

Valid in: CREATE INDEX Statement

Category: User Control of Index Usage

Restriction: This table option is not supported on the CAS server.

Data source: SPD Engine data set

Syntax

ASYNCINDEX= YES | NO

Arguments

YES
creates the indexes in parallel (asynchronously).

NO
creates one index at a time (synchronously). This is the default value.

Details

The SPD Engine can create multiple indexes for a data set at the same time. The SPD Engine spawns a single thread for each index created, and then processes the threads simultaneously. Although creating indexes in parallel is much faster than creating one index at a time, the default for this option is NO. Parallel creation requires additional utility work space and additional memory, which might not be available. If the index creation fails due to insufficient resources, you can do one of the following:

- set the SAS system option to MEMSIZE=0
- increase the size of the utility file space using the SPDEUTILLOC= system option

You increase the memory space that is used for index sorting using the SPDEINDEXSORTSIZE= system option. If you specify to create indexes in parallel, specify a large-enough space using the SPDEUTILLOC= system option.

BL_ALLOW_READ_ACCESS= Table Option

Specifies that the original table data is still visible to readers during bulk load.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the “BULKOPTS= Table Option” on page 1028
Data source:	DB2 under UNIX and PC

Syntax

BL_ALLOW_READ_ACCESS= YES | NO

Arguments

YES
specifies that the original (unchanged) data in the table is still visible to readers while bulk load is in progress.

NO
specifies that readers cannot view the original data in the table while bulk load is in progress. This is the default value.

Details

For more information about using this option, see the SQLU_ALLOW_READ_ACCESS parameter in *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

BL_COPY_LOCATION= Table Option

Specifies the directory to which DB2 saves a copy of the loaded data.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	DB2 under UNIX and PC

Syntax

BL_COPY_LOCATION= *pathname*

Arguments

pathname
specifies the path where the loaded data is copied.

See Also

Table Options:

- "BL_RECOVERABLE= Table Option" on page 1019
- "BULKLOAD= Table Option" on page 1026

BL_CPU_PARALLELISM= Table Option

Specifies the number of processes or threads to use when building table objects.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028

Data source: DB2 under UNIX and PC

Syntax

BL_CPU_PARALLELISM= *number-of-processes-or-threads*

Arguments

number-of-processes-or-threads
specifies the number of processes or threads that the load utility uses to parse, convert, and format data records when building table objects.

Details

This option exploits intrapartition parallelism and significantly improves load performance. It is particularly useful when loading presorted data, because record order in the source data is preserved.

The maximum number that is allowed is 30. If the value is 0 or has not been specified, the load utility selects an intelligent default. This default is based on the number of available CPUs on the system at run time. If there is insufficient memory to support the specified value, the utility adjusts the value.

When BL_CPU_PARALLELISM is greater than 1, the flushing operations are asynchronous, permitting the loader to exploit the CPU. If tables include either LOB or LONG VARCHAR data, parallelism is not supported. The value is set to 1, regardless of the number of system CPUs or the specified value.

Although use of this parameter is not restricted to symmetric multiprocessor (SMP) hardware, you might not obtain any discernible performance benefit from using it in non-SMP environments.

For more information about using BL_CPU_PARALLELISM=, see the CPU_PARALLELISM parameter in *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

See Also

Table Options:

- "BL_DATA_BUFFER_SIZE= Table Option" on page 1007
- "BL_DISK_PARALLELISM= Table Option" on page 1009
- "BULKLOAD= Table Option" on page 1026

BL_DATA_BUFFER_SIZE= Table Option

Specifies the total amount of memory to allocate for the bulk load utility to use as a buffer for transferring data.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	DB2 under UNIX and PC

Syntax

BL_DATA_BUFFER_SIZE= *buffer-size*

Arguments

buffer-size
specifies the total amount of memory (in 4KB pages) that is allocated for the bulk load utility to use as buffered space for transferring data within the utility. This setting does not consider the degree of parallelism that is available.

Details

If you specify a value that is less than the algorithmic minimum, the minimum required resource is used and no warning is returned. This memory is allocated directly from the utility heap, the size of which you can modify through the util_heap_sz database configuration parameter. If you do not specify a value, the utility calculates an intelligent default at run time. The calculation is based on a percentage of the free space that is available in the utility heap at the time of instantiation of the loader, as well as some characteristics of the table.

It is recommended that the buffer be several extents in size. An *extent* is the unit of movement for data within DB2, and the extent size can be one or more 4KB pages.

The DATA BUFFER parameter is useful when you are working with large objects (LOBs) because it reduces input and output waiting time. The data buffer is

allocated from the utility heap. Depending on the amount of storage that is available on your system, you should consider allocating more memory for use by the DB2 utilities. You can modify the database configuration parameter util_heap_sz accordingly. The default value for the Utility Heap Size configuration parameter is 5,000 4KB pages. Because load is one of several utilities that use memory from the utility heap, it is recommended that no more than 50% of the pages that are defined by this parameter be made available for the load utility.

For more information about using this option, see the DATA BUFFER parameter in *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

See Also

Table Options:

- "BL_CPU_PARALLELISM= Table Option" on page 1006.
- "BL_DISK_PARALLELISM= Table Option" on page 1009
- "BULKLOAD= Table Option" on page 1026

BL_DEFAULT_DIR= Table Option

Specifies the location where bulk load creates all intermediate files.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	Oracle

Syntax

BL_DEFAULT_DIR= *host-specific-directory-path*

Arguments

host-specific-directory-path
specifies the host-specific directory path where intermediate bulk-load files are created. The default is the current directory.

Details

The value that you specify for this option is prepended to the filename. Be sure to provide the complete, host-specific directory path, including the file and directory separator character to accommodate all platforms.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

BL_DISK_PARALLELISM= Table Option

Specifies the number of processes or threads to use when writing data to disk.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	DB2 under UNIX and PC

Syntax

BL_DISK_PARALLELISM= *number-of-processes-or-threads*

Arguments

number-of-processes-or-threads
specifies the number of processes or threads that the load utility uses to write data records to the table-space containers.

Details

This option exploits the available containers when it loads data and significantly improves load performance.

The maximum number that is allowed is the greater of four times the BL_CPU_PARALLELISM value, which the load utility actually uses, or 50. By default, BL_DISK_PARALLELISM is equal to the sum of the table-space containers on all table spaces that contain objects for the table that is being loaded except where this value exceeds the maximum number that is allowed.

If you do not specify a value, the utility selects an intelligent default that is based on the number of table-space containers and the characteristics of the table.

For more information about using this option, see the DISK_PARALLELISM parameter in *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

See Also

Table Options:

- "BL_CPU_PARALLELISM= Table Option" on page 1006
- "BULKLOAD= Table Option" on page 1026

BL_ERRORS= Table Option

Specifies that, after the indicated number of errors is received, the load should stop.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	Oracle

Syntax

BL_ERRORS= *number*

Arguments

number
specifies the number of errors that should be received before the load stops. The default is 1000000 errors.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

BL_EXCEPTION= Table Option

Specifies the exception table into which rows in error are copied.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028

Data source: DB2 under UNIX and PC

Syntax

BL_EXCEPTION= *exception-table-name*

Arguments

exception-table-name
specifies the exception table into which rows in error are copied.

Details

Any row that is in violation of a unique index or a primary key index is copied. DATALINK exceptions are also captured in the exception table. If you specify an unqualified table name, the table is qualified with the CURRENT SCHEMA. Information that is written to the exception table is not written to the dump file. In a partitioned database environment, you must define an exception table for those partitions on which the loading table is defined. However, the dump file contains rows that cannot be loaded because they are not valid or contain syntax errors.

For more information about using this option, see the FOR EXCEPTION parameter in *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*. For more information about the load exception table, see the load exception table topics in *IBM DB2 Universal Database Data Movement Utilities Guide and Reference* and *IBM DB2 Universal Database SQL Reference, Volume 1*.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

BL_INDEXING_MODE= Table Option

Specifies which scheme the DB2 load utility should use for index maintenance.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	DB2 under UNIX and PC

Syntax

BL_INDEXING_MODE= AUTOSELECT | REBUILD | INCREMENTAL | DEFERRED

Arguments

AUTOSELECT
specifies that the load utility automatically decides between REBUILD or INCREMENTAL mode.

REBUILD
specifies that all indexes are rebuilt.

INCREMENTAL
specifies that indexes are extended with new data.

DEFERRED
specifies that the load utility does not attempt index creation. Indexes are marked as needing a refresh.

Details

For more information about using the values for this option, see *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

BL_LOAD= Table Option

Specifies that, after the indicated number of rows is loaded, the load should stop.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028

Data source: Oracle

Syntax

BL_LOAD= *number-of-rows*

Arguments

number-of-rows
specifies the number of rows that should be loaded. The first rows from the table will be loaded. The default is to load all rows.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

BL_LOAD_REPLACE= Table Option

Specifies whether DB2 appends or replaces rows during bulk loading.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028

Data source: DB2 under UNIX and PC

Syntax

BL_LOAD_REPLACE= NO | YES

Arguments

NO
specifies that the CLI LOAD interface appends new rows of data to the DB2 table. This is the default value.

YES
specifies that the CLI LOAD interface replaces the existing data in the table.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

BL_LOG= Table Option

Identifies a log file that contains information such as statistics and error information for a bulk load.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	DB2 under UNIX and PC, Oracle

Syntax

BL_LOG= "*path-and-log-file-name***"**

Arguments

path-and-log-file-name
is a file to which information about the loading process is written. The default path and log filename is DBMS-specific.

Details

When the DBMS bulk load facility is invoked, it creates a log file. The contents of the log file are DBMS-specific. The BL_ prefix distinguishes this log file from the one created by the SAS log. If the BL_LOG= table option is specified with the same path and filename as an existing log, the new log replaces the existing log.

If the BL_LOG= table option is not specified, the log file is deleted automatically after a successful operation.

Example

The BL_LOG= table option is specified within the BL_BULKOPTS= table option:

```
bulkload=yes; bulkopts=(bl_log="c:\temp\bulkload.log");
```

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

BL_LOGFILE= Table Option

Specifies the filename for the bulk load log file.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	Oracle

Syntax

BL_LOGFILE= '*log-file-name*'

Arguments

log-file-name
specifies a name for the bulk load log file. The default is a generated filename that has the template BL_*TablenameUniquenumber*.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

BL_OPTIONS= Table Option

Passes options to the DBMS bulk load facility, affecting how it loads and processes data.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	DB2 under UNIX and PC

Syntax

BL_OPTIONS= '*option* [, …*option*] '

Arguments

option
specifies a valid DB2 option. By default, no options are passed.

Details

The BL_OPTIONS= table option enables you to pass options to the DBMS bulk load facility when it is invoked, thereby affecting how data is loaded and processed. You must separate multiple options with commas and enclose the entire string of options in single quotation marks.

This option passes DB2 file type modifiers to DB2 LOAD or IMPORT commands to affect how data is loaded and processed. Not all DB2 file type modifiers are appropriate for all situations. You can specify one or more DB2 file type modifiers with .IXF files. For a list of file type modifiers, see the description of the LOAD and IMPORT utilities in *DB2 Data Movement Utilities Guide and Reference*.

Example

This option is specified within the BULKOPTS= table option:

```
bulkload=yes; bulkopts=(bl_options='option1, option2');
```

See Also

Table Options:

- “BULKLOAD= Table Option” on page 1026
- “BULKOPTS= Table Option” on page 1028

BL_PARALLEL Table Option

Specifies whether to perform a parallel bulk load.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the “BULKOPTS= Table Option” on page 1028

Data source: Oracle

Syntax

BL_PARALLEL= YES | NO

Arguments

YES
specifies that a parallel load should be performed.

NO
specifies that a parallel load is not performed. That is the default behavior.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 1026

BL_PORT_MAX= Table Option

Sets the highest available port number for concurrent uploads.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the “BULKOPTS= Table Option” on page 1028

Data source: DB2 under UNIX and PC

Syntax

BL_PORT_MAX= *integer*

Arguments

integer
specifies a positive integer that represents the highest available port number for concurrent uploads.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

BL_PORT_MIN= Table Option

Sets the lowest available port number for concurrent uploads.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	DB2 under UNIX and PC

Syntax

BL_PORT_MIN= *integer*

Arguments

integer
specifies a positive integer that represents the lowest available port number for concurrent uploads.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

BL_RECOVERABLE= Table Option

Specifies whether the LOAD process is recoverable.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	DB2 under UNIX and PC, Oracle

Syntax

BL_RECOVERABLE= YES | NO

Arguments

YES
specifies that the LOAD process is recoverable. For DB2, YES also specifies that BL_COPY_LOCATION= should specify the copy location for the data.

NO
specifies that the LOAD process is not recoverable.

Details

DB2 under UNIX and PC Hosts: The default is NO.

Oracle: The default is YES. Set this option to NO to improve direct load performance. Specifying NO adds the UNRECOVERABLE keyword before the LOAD keyword in the control file.

CAUTION
Be aware that an unrecoverable load does not log loaded data into the redo log file. Therefore, media recovery is disabled for the loaded table. For more information about the implications of using the UNRECOVERABLE parameter in Oracle, see your Oracle utilities documentation.

See Also

Table Options:

- "BL_COPY_LOCATION= Table Option" on page 1005
- "BULKLOAD= Table Option" on page 1026

BL_REMOTE_FILE= Table Option

Specifies the base filename and location of DB2 LOAD temporary files.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	DB2 under UNIX and PC

Syntax

BL_REMOTE_FILE= *pathname-and-base-filename*

Arguments

pathname-and-base-filename

specifies the full pathname and base filename to which DB2 appends extensions (such as .log, .msg and .dat files) to create temporary files during load operations. By default, BL_<*table*>_<*unique-ID*> is the form of the base filename.

table

specifies the table name.

unique-ID

specifies a number that prevents collisions in the event of two or more simultaneous bulk loads of a particular table. The table driver generates this number.

Details

When you specify this option, the DB2 LOAD command is used (instead of the IMPORT command).

For *pathname*, specify a location on a DB2 server that is accessed exclusively by a single DB2 server instance, and for which the instance owner has Read and Write permissions. Make sure that each LOAD command is associated with a unique *pathname-and-base-filename* value.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

BL_SKIP= Table Option

Specifies to skip the indicated number of rows before starting the bulk load.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028

Data source: Oracle

Syntax

BL_SKIP= *number-of-rows*

Arguments

number-of-rows
specifies the number of rows to skip before beginning the load. The default is 0, which means that no records are skipped.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

BL_SKIP_INDEX_MAINTENANCE= Table Option

Specifies whether to perform index maintenance on the bulk load.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the"BULKOPTS= Table Option" on page 1028

Data source: Oracle

Syntax

BL_SKIP_INDEX_MAINTENANCE= YES | NO

Arguments

YES
specifies to stop index maintenance on the load. This causes the index partitions that would have had index keys added to them to be marked Index Unusable. The index segment is inconsistent with the data it indexes. Index segments that are not affected by the load retain the Index Unusable state that they had prior to the load.

NO
specifies that index maintenance is performed on the load. This is the default value.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

BL_SKIP_UNUSABLE_INDEXES= Table Option

Specifies whether to skip index entries that are in an unusable state and continue with the bulk load.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	Oracle

Syntax

BL_SKIP_UNUSABLE_INDEXES= YES | NO

Arguments

YES
specifies that the unusable index entry should be skipped. This is the default value.

NO
specifies that the unusable index entry should not be skipped.

Details

If an index in an Index Unusable state is encountered, by default, it is skipped and the load operation continues. This allows the SQL*Loader to load a table with indexes that are in an unusable state before beginning the load. Indexes that are not in an unusable state at load time are maintained by the SQL*Loader. Indexes that are in an unusable state at load time are not maintained and remain in an unusable state at load completion.

If this bulk load option is not specified, the default value is specified in the Oracle database configuration parameter, SKIP_UNUSABLE_INDEXES. This value is specified in the initialization parameter file. The BL_SKIP_UNUSABLE_INDEXES bulk load table option overrides the value of the SKIP_UNUSABLE_INDEXES configuration parameter in the initialization parameter file.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 1026

BL_WARNING_COUNT= Table Option

Specifies the maximum number of row warnings to allow before the load fails.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the “BULKOPTS= Table Option” on page 1028
Data source:	DB2 under UNIX and PC

Syntax

BL_WARNING_COUNT= *number-of-warnings*

Arguments

number-of-warnings
specifies the maximum number of row warnings to allow before the load fails.

Details

To specify this option, you must first set BULKLOAD=YES and also specify a value for BL_REMOTE_FILE=.

Use this option to limit the maximum number of rows that generate warnings. See the log file for information about why the rows generated warnings.

See Also

Table Options:

- "BL_REMOTE_FILE= Table Option" on page 1020
- "BULKLOAD= Table Option" on page 1026

BUFNO= Table Option

Specifies the number of buffers to be allocated for processing a SAS data set.

Category:	Table Control
Restriction:	This table option is not supported on the CAS server.
Data source:	SAS data set

Syntax

BUFNO= *n* | *n*K | *hex*X | MIN | MAX

Arguments

***n* | *n*K**

specifies the number of buffers in multiples of 1 (bytes); 1,024 (kilobytes). For example, a value of `8` specifies 8 buffers, and a value of `1K` specifies 1024 buffers.

Requirement *K* must be uppercased.

***hex*X**

specifies the number of buffers as a hexadecimal value. You must specify the value beginning with a number (0–9), followed by an X. For example, the value `2dX` sets the number of buffers to 45 buffers.

Requirement *X* must be uppercased.

MIN
sets the minimum number of buffers to 0, which causes SAS to use the minimum optimal value for the operating environment. This is the default value.

MAX
sets the number of buffers to the maximum possible number in your operating environment, up to the largest four-byte, signed integer. The largest four-byte, signed integer is 2^{31}–1, or approximately 2 billion.

Details

The buffer number is not a permanent attribute of the data set; it is valid only for the current operation.

The BUFNO= table option applies to SAS data sets that are opened for input, output, or update.

A larger number of buffers can speed up execution time by limiting the number of input and output (I/O) operations that are required for a particular SAS data set. However, the improvement in execution time comes at the expense of increased memory consumption.

To reduce I/O operations on a small data set as well as speed execution time, allocate one buffer for each page of data to be processed. This technique is most effective if you read the same observations several times during processing.

BUFSIZE= Table Option

Specifies the size of a permanent buffer page for an output SAS data set.

Category:	Table Control
Restrictions:	This table option is not supported on the CAS server. Use with output data sets only.
Data source:	SAS data set

Syntax

BUFSIZE=*n* | *n*K | *n*M | *n*G | *hex*X | MIN | MAX

Arguments

***n* | *n*K | *n*M | *n*G**
specifies the page size in multiples of 1 (bytes); 1,024 (kilobytes); 1,048,576 (megabytes); or 1,073,741,824 (gigabytes). For example, a value of `8` specifies a page size of 8 bytes, and a value of `4k` specifies a page size of 4096 bytes.

Requirement *K*, *M*, and *G* must be uppercased.

***hex*X**
specifies the page size as a hexadecimal value. You must specify the value beginning with a number (0–9), followed by an X. For example, the value `2dx` sets the page size to 45 bytes.

Requirement *X* must be uppercased.

MIN
sets the minimum number of buffers to 0, which causes SAS to use the minimum optimal value for the operating environment.

MAX
sets the page size to the maximum possible number in your operating environment, up to the largest four-byte, signed integer. The largest four-byte, signed integer is 2^{31}–1, or approximately 2 billion bytes.

Details

The page size is the amount of data that can be transferred for a single I/O operation to one buffer. The page size is a permanent attribute of the data set and is used when the data set is processed.

A larger page size can speed up execution time by reducing the number of times SAS has to read from or write to the storage medium. However, the improvement in execution time comes at the cost of increased memory consumption.

To change the page size, copy the data set and either specify a new page or use the SAS default. To reset the page size to the default value in your operating environment, specify BUFSIZE=0.

Operating Environment Information: The default value for the BUFSIZE= table option is determined by your operating environment and is set to optimize sequential access. To improve performance for direct (random) access, you should change the value for BUFSIZE=.

BULKLOAD= Table Option

Determines whether SAS uses a DBMS facility to insert data into a DBMS table.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Interaction:	Used in conjunction with "BULKOPTS= Table Option" on page 1028.
Data source:	DB2 under UNIX and PC, Google BigQuery, Impala, MDS, Oracle, Teradata

Syntax

BULKLOAD= YES | NO

Arguments

YES
calls a DBMS-specific bulk-load facility in order to insert or append rows to a DBMS table.

NO
does not call the DBMS-specific bulk-load facility. This is the default value.

Details

Overview

Using BULKLOAD=YES is the fastest way to insert rows into a DBMS table.

You can specify data source-specific options in the BULKOPTS= table option (BL_BULKOPTS= for Oracle). BULKOPTS= functions as a container for the data source-specific options. For more information, see "BULKOPTS= Table Option" on page 1028.

When the BULKLOAD= table option is not set, a simple multi-row insert SQL scheme is used to insert data rows.

Usage Notes

Google BigQuery

Bulk-loading options for Google BigQuery apply to Insert operations only. Bulk loading can be enabled in the LIBNAME statement for the BIGQUERY engine or using the BULKLOAD= and BULKOPTS= table options. When the options are specified both ways, the table option overrides the LIBNAME option. See "Bulk Loading for Google BigQuery" in *SAS/ACCESS for Relational Databases: Reference* for valid Google BigQuery bulk-load options.

Impala

Bulk loading to the Impala server can be accomplished in two ways: you can use the WebHDFS interface to Hadoop to push data to HDFS, or you can configure a required set of Hadoop JAR files. Both approaches require Hadoop configuration files needed by SAS to be in one location and available to the client machine. To use WebHDFS, you must additionally set the SAS_HADOOP_RESTFUL= environment variable to 1. To use Java, you must make the Hadoop JAR location known to the client machine and ensure that the SAS_HADOOP_RESTFUL= environment variable is not set to 1 (or TRUE or YES).

Specifying BULKLOAD=YES causes two CREATE TABLE statements to be issued to the Impala server. One creates the target Impala table. The other creates a temporary table. SAS uses WebHDFS to upload table data to the HDFS /tmp directory. The resulting file is a UTF-8 delimited text file. SAS issues a LOAD DATA statement to move the data file from the /tmp directory to the temporary table, and then issues an INSERT INTO statement that copies and transforms the text data

from the temporary table to the target table. The temporary table is then deleted from HDFS.

MDS

BULKLOAD= is available for insert operations only. When the BULKLOAD= table option is set, newly inserted rows are committed immediately and become visible to existing transactions. When the BULKLOAD= table option is not set, newly inserted rows are not visible until the existing transactions are committed or rolled back.

Teradata

Specifying BULKLOAD=YES invokes the Teradata Parallel Transporter (TPT) API protocol driver. The default TPT operator is the Stream operator. Use the BULKOPTS= table option to specify a different TPT operator.

Example

```
create table mylib.myexample {options BULKLOAD=YES
               BULKOPTS=(BL_DELETE_DATAFILE=NO)}(col1 float);
```

See Also

Concepts:

- “FedSQL Statement Table Options by Data Source” on page 991

Table Options:

- “BULKOPTS= Table Option” on page 1028

BULKOPTS= Table Option

Provides a container for bulk-loading options.

Category:	Bulk Loading
Alias:	In Oracle, this option has the name BL_BULKOPTS=
Restriction:	This table option is not supported on the CAS server.
Requirement:	BULKLOAD=YES must be set in order to use BULKOPTS=
Data source:	DB2 under UNIX and PC, Google BigQuery, Impala, Oracle, Teradata

Syntax

BULKOPTS= (*option*[…*option*])

Arguments

option[...option]
specifies one or more bulk load table options separated by spaces.

Details

To specify BULKOPTS= table option, you must first specify BULKLOAD=YES. For more information, see "BULKLOAD= Table Option" on page 1026.

The BULKOPTS= table option is a container option that is required in order to specify other bulk load table options.

Bulk load options are data source-specific. See "FedSQL Statement Table Options by Data Source" on page 991 for a list of the options that can be specified in this container for each data source.

Google BigQuery users: FedSQL supports the same bulk-load options as the SAS/ACCESS to Google BigQuery LIBNAME engine. See "Bulk Loading for Google BigQuery" in *SAS/ACCESS for Relational Databases: Reference* for valid options.

Examples

Example 1

The following example uses the BULKLOAD=YES and BULKOPTS= table options to submit bulk-loading options to DB2 under UNIX and PC:

```
BULKLOAD=YES BULKOPTS=(BL_LOG='c:\temp\bulkload.log'  BL_LOAD_REPLACE=yes
      BL_OPTIONS='ERRORS=999, LOAD=2000');
```

Example 2

The following example uses the BULKLOAD=YES and BL_BULKOPTS= options to submit bulk-loading options to Oracle:

```
BULKLOAD=YES BL_BULKOPTS=(BL_DEFAULT_DIR='C:\mylogs' BL_LOGFILE='novemberdatalog'
BL_ERRORS=999)
```

Example 3

The following example uses the BULKLOAD=YES and BULKOPTS= options to submit bulk-loading options to Teradata. The Teradata Parallel Transporter (TPT) is used to load data into Teradata tables when BULKLOAD=YES. The TPT job uses the Stream operator by default. In the example below, the TD_TPT_OPER= table option specifies to use the Load operator instead, among other things.

```
BULKLOAD=YES BULKOPTS=(TD_TPT_OPER=LOAD TD_ERROR_LIMIT=1
TD_TRACE_OUTPUT=tpttrace TD_TRACE_LEVEL=TD_OPER_ALL
TD_MAX_SESSIONS=8 TD_TRACE_LEVEL_INF=TD_OPER_ALL)
```

Example 4

The following example uses the BULKLOAD=YES and BULKOPTS= options to submit bulk-loading options to Impala. The bulk load request will use WebHDFS to load the data to HDFS. Note that the SAS_HADOOP_RESTFUL= environment variable must also be set to 1 for the request to succeed.

```
BULKLOAD=YES BULKOPTS=(USER=HDFS PASS="hdfs-pwd"
CONFIG="my-config-path")
```

Example 5

The following example uses the BULKLOAD=YES and BULKOPTS= options to submit bulk-load options to Google BigQuery. The Google BigQuery loader client application is used to move data from the client to the Google BigQuery database. Valid bulk-load options are documented in “Bulk Loading Data in Google BigQuery” in *SAS/ACCESS for Relational Databases: Reference*.

```
BULKLOAD=YES BULKOPTS=(BL_DELETE_DATA_FILE=NO;BL_DEFAULT_DIR='/u/myid/temp')
```

See Also

Table Options:

- “TD_TPT_OPER= Table Option” on page 1090
- “CONFIG= Table Option” on page 1032

COMPRESS= Table Option

Specifies how rows are compressed in a new output data set or table.

Category: Table Control

Restrictions: This table option is not supported on the CAS server.

Use with output tables only.

Data source: SAS data set, SPD Engine data set, SPD Server table

Syntax

COMPRESS= NO | YES | CHAR | BINARY

Arguments

CHAR | YES
specifies that the rows in a newly created table are compressed (variable length records). SAS data sets are compressed. by using Run Length Encoding (RLE). RLE compresses rows by reducing repeated consecutive characters (including blanks) to two-byte or three-byte representations. SPD Engine data sets are compressed by using the run length compression algorithm SPDSRLC2. SPD Server tables are compressed by using the run-length compression algorithm SPDSRLLC.

Alias ON (SPD Engine data set only)

Tip Use this compression algorithm for character data.

BINARY
specifies that the rows in a newly created data set or table are compressed by SAS using Ross Data Compression (RDC). RDC combines run-length encoding and sliding-window compression to compress the file. SPD Engine data sets are compressed by the SPD Engine by using SPDSRDC.

Tip This method is highly effective for compressing medium to large (several hundred bytes or larger) blocks of binary data (numeric variables). Because the compression function operates on a single record at a time, the record length needs to be several hundred bytes or larger for effective compression.

NO
specifies that the rows in a newly created table are uncompressed (fixed-length records).

Details

Compressing a table is a process that reduces the number of bytes required to represent each row. Advantages of compressing a table include reduced storage requirements for the table and fewer I/O operations necessary to read or write to the data during processing. However, more CPU resources are required to read a compressed table (because of the overhead of uncompressing each row). Also, there are situations where the resulting file size might increase rather than decrease.

Use the COMPRESS= table option to compress an individual table. Specify the option for an output table only—that is, a table name on the CREATE TABLE statement.

Note: In SPD Engine data sets and SPD Server tables, encryption and compression are mutually exclusive. You cannot specify encryption for a compressed SPD Engine data set or SPD Server table. You cannot compress an encrypted SPD Engine data set or SPD Server table.

After a table is compressed, the setting is a permanent attribute of the table, which means that to change the setting, you must re-create the table. That is, to uncompress a table, you must drop the table by using the DROP TABLE statement and re-create the table with the CREATE TABLE statement and the COMPRESS=NO option.

When you specify COMPRESS= to compress an SPD Engine data set or SPD Server table, the table drivers compress, by blocks, the table as it is created. To specify the size of the compressed blocks, use the "IOBLOCKSIZE= Table Option" on page 1051.

The SPD Engine table driver also supports the PADCOMPRESS= table option when creating or updating the SPD Engine data set. The PADCOMPRESS= option enables you to add padding to the newly compressed blocks. See the "PADCOMPRESS= Table Option" on page 1058.

Comparisons

The COMPRESS= table option overrides the COMPRESS= connection string option.

SAS data sets only: When you create a compressed table, you can also specify the REUSE=YES table option in order to track and reuse space. With REUSE=YES, new rows are inserted in space freed when other rows are updated or deleted. When the default REUSE=NO is in effect, new rows are appended to the existing table.

See Also

Table Options:

- "ENCRYPT= Table Option" on page 1039
- "POINTOBS= Table Option" on page 1062
- "IOBLOCKSIZE= Table Option" on page 1051
- "PADCOMPRESS= Table Option" on page 1058
- "REUSE= Table Option" on page 1068

CONFIG= Table Option

Specifies a file or pathname for Hadoop configuration path resolution.

Category: Bulk Loading

Alias: CFG=, HD_CONFIG=, CONFIGDIR=, CFGDIR=, HD_CONFIGDIR=

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028.

Data source: Impala

Syntax

CONFIG="*configuration-path*"

Arguments

"*configuration-path*"
specifies the name of a single file or a directory that contains Hadoop configuration information that is required by SAS on your machine. If this option is not specified, SAS searches for the value in the SAS_HADOOP_CONFIG_PATH environment variable.

Details

The SAS_HADOOP_CONFIG_PATH environment variable can list multiple directories (defined similarly to a PATH variable) to search for configuration information. Use the CONFIG= option when you want to use a specific configuration file or to search a specific directory for configuration information.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

DATAFILE= Table Option

Specifies an alternate name and location for the temporary HDFS file.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028.

Data source: Impala

Syntax

DATAFILE= "*filename*"

Arguments

"*filename*"
specifies the full pathname to a file. When this option is not specified, the default pathname for the temporary file is "/tmp/bl_*tablename*_*unixtimeval*.dat".

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

DBCREATE_INDEX_OPTS= Table Option

Allows DBMS-specific options to be added to the CREATE INDEX statement.

Category:	Index Control
Restriction:	This table option is not supported on the CAS server.
Data source:	Microsoft SQL Server, ODBC

Syntax

DBCREATE_INDEX_OPTS= '*DBMS-SQL-clauses*'

Arguments

DBMS-SQL-clauses
specifies one or more DBMS-specific clauses that can be appended to the end of an SQL CREATE INDEX statement.

Details

This option enables you to add DBMS-specific clauses to the end of the CREATE INDEX statement. The interface passes the CREATE INDEX statement and its clauses to the DBMS, which executes the statement and creates the DBMS index.

Example

In the following example, a Hive index is created with the value of the DBCREATE_INDEX_OPTS= “as 'compact' with deferred rebuild” option appended to the CREATE INDEX statement:

```
create index "COL1_1A03" on "TKTS002_1A03"{options
DBCREATE_INDEX_OPTS="as 'compact'
with deferred rebuild"}("COL1")
```

The following CREATE INDEX statement is passed to the DBMS in order to create the index:

```
create index 'COL1_1A03' on table 'TKTS002_1A03'('COL1') as 'compact'
with
deferred rebuild
```

DBCREATE_TABLE_OPTS= Table Option

Allows database-specific options to be placed after the CREATE TABLE statement in generated SQL code.

Category:	Table Control
Alias:	POST_STMT_OPTS= (Hive only, beginning with the SAS 9.4M5)
Restriction:	This table option is not supported on the CAS server.
Data source:	Amazon Redshift, DB2 under UNIX and PC, Google BigQuery, Greenplum, HAWQ, Hive, Microsoft SQL Server, MySQL, Netezza, Oracle, SAP HANA, Spark, Teradata

Syntax

DBCREATE_TABLE_OPTS= '*DBMS-SQL-clauses*'

Arguments

DBMS-SQL-options
specifies one or more DBMS-specific options that can be appended to the end of an SQL CREATE TABLE statement.

Details

This option enables you to add DBMS-specific options to the end of the CREATE TABLE statement. The interface passes the CREATE TABLE statement and its options to the DBMS, which executes the statement and creates the DBMS table.

The availability and behavior of FedSQL statement options are data-source specific. If DBCREATE_TABLE_OPTS= options are used in a statement other than CREATE TABLE, those options might be ignored.

Examples

Example 1: Google BigQuery

In the following example, the Google BigQuery table TEMP is created with the description “table created by SAS” appended to the CREATE TABLE statement:

```
create table a."my_dataset".temp {option dbcreate_table_opts='options
(description="table created by SAS")'}(col1 char(8));
```

The following CREATE TABLE statement is sent to the DBMS:

```
create table `my_project`.`my_dataset`.`TEMP` (`COL1` STRING)
 options  (description="table created by SAS")
```

Example 2: Greenplum

In the following example, the Greenplum table TEMP is created with the value of the DBCREATE_TABLE_OPTS='DISTRIBUTED BY ("B")' option appended to the CREATE TABLE statement:

```
create table temp {options dbcreate_table_opts='distributed by ("b")'}
(a int, b int);
```

The following CREATE TABLE statement is passed to the DBMS in order to create the table:

```
create table temp (a int, b int) distributed by ("b")
```

Example 3: HAWQ

In the following example, the HAWQ table TEMP is created with the value of the DBCREATE_TABLE_OPTS='DISTRIBUTED BY ("A")' option appended to the CREATE TABLE statement:

```
create table temp {options DBCREATE_TABLE_OPTS='DISTRIBUTED by ("A")'}(a int, b int);
```

The following CREATE TABLE statement is passed to the DBMS in order to create the table:

```
create table temp {options DBCREATE_TABLE_OPTS='DISTRIBUTED by ("A")'}(a int, b int);
```

Example 4: Hive

In the following example, the Hive table TEMP is created with the value of the DBCREATE_TABLE_OPTS='PARTITIONED BY ("col1")' option appended to the CREATE TABLE statement:

```
create table temp {options DBCREATE_TABLE_OPTS='PARTITIONED BY (col1
int)'} (col1 int, col2 int);
```

The following CREATE TABLE statement is passed to the DBMS in order to create the table:

```
create table `temp` (`COL2` int) PARTITIONED BY (col1 int)
```

In Hive, beginning in 9.4M5, you can use DBCREATE_TABLE_OPTS= in combination with these related table options: PRE_TABLE_OPTS= and

POST_TABLE_OPTS=. For usage information, see "PRE_TABLE_OPTS= Table Option" on page 1065 and "POST_TABLE_OPTS= Table Option" on page 1064. Beginning in 9.4M5, you can also use the alias POST_STMT_OPTS= instead of DBCREATE_TABLE_OPTS=. Do not specify POST_STMT_OPTS= and DBCREATE_TABLE_OPTS= in the same statement. Specifying both names in the same statement results in an error.

Example 5: MySQL

In the following example, the MySQL table TEMP is created with the value of the DBCREATE_TABLE_OPTS='PARTITIONING KEY (x) USING HASHING' option appended to the CREATE TABLE statement:

```
create table temp {option dbcreate_table_opts='partitioning key (x) using hashing'}
   (x double);
```

The following CREATE TABLE statement is passed to the DBMS in order to create the table:

```
create table temp (x double) partitioning key (x) using hashing;
```

Example 6: Netezza

In the following example, the Netezza table TEMP is created with the value of the DBCREATE_TABLE_OPTS='DISTRIBUTE ON (b)' option appended to the CREATE TABLE statement:

```
create table temp {options dbcreate_table_opts='distribute on (b)'}
(a int, b int);
```

The following CREATE TABLE statement is passed to the DBMS in order to create the table:

```
create table temp (A int, B int) distribute on (b)
```

Example 7: Oracle

In the following example, the Oracle table TEMP is created with the value of the DBCREATE_TABLE_OPTS='NOLOGGING' option appended to the CREATE TABLE statement:

```
create table temp{options dbcreate_table_opts='nologging'}
(a int, b int)
```

The following CREATE TABLE statement is passed to the DBMS in order to create the table:

```
create table temp ("a" number (10) , "b" number (10) ) nologging
```

Example 8: SAP HANA

In the following example, the SAP HANA table TEMP is created with the value of the DBCREATE_TABLE_OPTS='PARTITION BY RANGE' option appended to the CREATE TABLE statement:

```
create table temp {options table_type=column dbcreate_table_opts="partition by range (x)
(partition 1 <= values < 10, partition others)"}
```

```
(x int, y int);
```

The following CREATE TABLE statement is passed to the DBMS in order to create the table:

```
create column table temp (x int, y int) partition by range (x)
(partition 1 <= values < 10, partition others)
```

Example 9: Spark

In the following example, the Spark table TEMP is created with the value of the DBCREATE_TABLE_OPTS='PARTITIONED BY ("B")' option appended to the CREATE TABLE statement:

```
create table temp {options dbcreate_table_opts='PARTITIONED by (b int)'}
 (a int, b int);
```

The following CREATE TABLE statement is passed to the DBMS in order to create the table:

```
create table temp (a int) PARTITIONED by (b int)
```

Example 10: Teradata

In the following example, the Teradata table TEMP is created with the value of the DBCREATE_TABLE_OPTS='PRIMARY INDEX (B)' option appended to the CREATE TABLE statement:

```
create table temp {options dbcreate_table_opts='primary index(b)'}
(a int, b int);
```

The following CREATE TABLE statement is passed to the DBMS in order to create the table:

```
create table temp (a int, b int) primary index(b)
```

See Also

Table Options:

- “POST_TABLE_OPTS= Table Option” on page 1064
- “PRE_TABLE_OPTS= Table Option” on page 1065
- “TABLE_TYPE= Table Option” on page 1070

ENCODING= Table Option

Specifies the encoding to use when creating a SAS data set.

Category:	Table Control
Restriction:	This table option is not supported on the CAS server.
Data source:	SAS data set

Syntax

ENCODING=*encoding-value*

Arguments

encoding-value
specifies an encoding value.

See "Encoding for NLS" in *SAS National Language Support (NLS): Reference Guide*

Details

SAS data sets are created with the session encoding by default. You can specify a different encoding for a new table by using the ENCODING= table option. ENCODING= is supported for input only. You cannot change the encoding of an existing table temporarily by specifying ENCODING= in the SELECT statement. Likewise, you cannot change the encoding of an existing table permanently by specifying it in the UPDATE statement.

Specify the ENCODING= table option in the CREATE TABLE statement before the columns are defined. Here is an example:

```
create table test {options encoding=ebcdic1145} (x char(8));
```

The ENCODING= table option is not supported with CREATE TABLE AS. If you specify the table option with CREATE TABLE AS, you get an error. The requested table will be created, but it will use the session encoding. However, if you use CREATE TABLE AS to insert an existing table that has an encoding specified into a new table that has no encoding specified , then the new table will have the encoding of the inserted table. For example, the following request creates table Test2 with the encoding ebcdic1145 Spain (EBCDIC), which is the encoding used by table Test.

```
create table test2 as select * from test;
```

ENCRYPT= Table Option

Specifies whether to encrypt an output SAS data set.

Category:	Table Control
Restrictions:	Use with output tables only. This table option is not supported on the CAS server. In SPD Engine data sets, this table option cannot be used with COMPRESS=.
Data source:	SAS data set, SPD Engine data set, SPD Server table

Syntax

ENCRYPT= | | |

Arguments

AES or AES2
encrypts the file using AES (Advanced Encryption Standard) algorithm. AES provides encryption by using SAS/SECURE software, which is included with Base SAS software. You must specify the ENCRYPTKEY= table option when you are using ENCRYPT=AES or ENCRYPT=AES2, or have a recorded key in a metadata-bound library. For more information, see "ENCRYPTKEY= Table Option" on page 1042.

Restriction SPD Engine and SPD Server do not support AES2 encryption.

Note AES2 encryption is supported beginning with SAS 9.4M5.

CAUTION **Record all ENCRYPTKEY= values when you are using ENCRYPT=AES or ENCRYPT=AES2.** If you forget to record the ENCRYPTKEY= value, you lose your data. SAS cannot assist you in recovering the ENCRYPTKEY= value. The following note is written to the log:

```
Note: If you lose or forget the ENCRYPTKEY= value, there will
be no way to open the file or recover the data.
```

NO
does not encrypt the data set.

YES
encrypts the data set using the SASProprietary algorithm. This encryption method uses passwords that are stored in the data set. At a minimum, you must specify the READ= or the PW= table option at the same time that you specify ENCRYPT=YES. For more information, see "Example: Using the ENCRYPT=YES Option" on page 1041.

Note Because the encryption method uses passwords, you cannot change *any* password on an encrypted file without re-creating the table.

CAUTION **Record all passwords when using ENCRYPT=YES.** If you forget the passwords, you cannot reset them. This is a time-consuming and resource-intensive process.

Details

When you use ENCRYPT=YES, the following rules apply:

- If the data file is encrypted, all associated indexes are also encrypted, except for SPD Server tables. SPD Server does not encrypt table indexes or metadata. Only table row data are encrypted.
- In order to copy an encrypted data file, the output engine must support encryption. Otherwise, the data file is not copied.

- You cannot use PROC CPORT on SAS Proprietary encrypted data files.

When you use ENCRYPT=AES or ENCRYPT=AES2, the following rules apply:

- You must have SAS/SECURE software.
- You must use the ENCRYPTKEY= table option when encrypting a table.
- You must use the ENCRYPTKEY= table option to enable decryption.
- A data set encrypted with AES encryption can be decrypted only by engines that support AES encryption.
- You cannot change the ENCRYPTKEY= value on an AES encrypted data file without re-creating the data file. The AUTHLIB procedure must be used to change the recorded key of a metadata-bound library.
- In Base SAS, data files with referential integrity constraints can use AES encryption. All primary key and foreign key data files must use the same encryption key that opens all referencing foreign key and primary key data files.

If a default Base SAS data set with AES2 encryption is copied to create a new SPD Engine data set or SPD Server table, the encryption converts to AES. A warning is written to the log.

For more information about the encryption provided by SAS, see *Encryption in SAS*.

Example: Using the ENCRYPT=YES Option

This example creates an encrypted data set using the SASProprietary algorithm:

```
create table myfiles.salary {option encrypt=yes read=green}
   (name char(15),
    yrsal double,
    bonuspct double);
insert into myfiles.salary {option read=green} values ('Muriel', 34567, 3.2);
insert into myfiles.salary {option read=green} values ('Bjorn', 74644, 2.5);
insert into myfiles.salary {option read=green} values ('Freda', 38755, 4.1);
insert into myfiles.salary {option read=green} values ('Benny', 29855, 3.5);
insert into myfiles.salary {option read=green} values ('Agnetha', 70998, 4.1);
```

To retrieve data from the data set, you must specify the Read password:

```
select * from myfiles.salary {option read=green};
```

See Also

Table Options:

- “ENCRYPTKEY= Table Option” on page 1042
- “PW= Table Option” on page 1066
- “READ= Table Option Table Option” on page 1067

ENCRYPTKEY= Table Option

Specifies a key value for AES and AES2 encryption.

Category: Table Control

Restrictions: Use only with AES and AES2 encrypted data files.
This table option is not supported on the CAS server.

Data source: SAS data set, SPD Engine data set, SPD Server table

Note: Check your log after this operation to ensure that the encrypt key values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in *SAS Programmer’s Guide: Essentials*.

Syntax

ENCRYPTKEY= [| ']*key-value* [| ']

Syntax Description

key-value

assigns an encrypt key value. You must specify the ENCRYPTKEY= table option when you are using ENCRYPT=AES or ENCRYPT=AES2, unless you are creating a data set in a metadata-bound library with a recorded key. The key value can be up to 64 bytes long. The ENCRYPTKEY= value can be created with or without quotation marks using the following rules:

no quotation marks

- alphanumeric characters and underscores only
- up to 64 bytes
- uppercase and lowercase letters
- must start with a letter
- cannot include blank spaces
- is not case-sensitive

Example `encryptkey=key_value_1234`

single quotation marks

- alphanumeric, special, and DBCS characters
- up to 64 bytes
- uppercase and lowercase letters
- can include blank spaces, but cannot contain all blanks
- is case-sensitive

Example `encryptkey='key-value'`

```
encryptkey='1234*#mykey'
```

double quotation marks

- alphanumeric, special, and DBCS characters
- up to 64 bytes
- uppercase and lowercase letters
- enables macro resolution
- can include blank spaces, but cannot contain all blanks
- is case-sensitive

Example

```
encryptkey="key-value"
encryptkey="1234*#mykey"

%let mykey=abcdefghi12;
encryptkey=&key-value
```

Note When the ENCRYPTKEY= key value uses DBCS characters, the 64-byte limit applies to the character string after it has been transcoded to UTF-8 encoding. You can use the following DATA step to calculate the length in bytes of a key value in DBCS:

```
data _null_;
    key=length(unicodec("key-value","UTF8"));
    put "key length=" key;
run;
```

Details

CAUTION

Record the key value. If you forget the ENCRYPTKEY= key value, you lose your data. SAS cannot assist you in recovering the ENCRYPTKEY= key value because the key value is not stored with the data set. The following warning is written to the log:

```
WARNING: If you lose or forget the ENCRYPTKEY= value, there
will be not be any way to open the file or recover the data.
```

You must use the ENCRYPTKEY= option when you are creating or accessing a SAS file with AES or AES2 encryption.

The ENCRYPTKEY= table option does not protect the file from deletion or replacement. Encrypted data sets can be deleted by using any of the following scenarios without having to specify an ENCRYPTKEY= key value:

- the KILL option in PROC DATASETS
- the DROP statement in PROC SQL
- the DELETE statement in PROC DATASETS or the DELETE procedure
- the DROP TABLE statement in FedSQL

The ENCRYPTKEY= option prevents access to the contents of the file only. To protect the file from deletion or replacement, the file must also contain an ALTER= password.

The following DATASETS procedure statements require you to specify the ENCRYPTKEY= key value when working with protected files: AGE, AUDIT, APPEND, CHANGE, CONTENTS, MODIFY, REBUILD, and REPAIR statements.

```
append base=name data=name(encryptkey=key-value);
run;
```

The option can be specified either in parentheses after the name of the SAS data file or after a forward slash.

It is possible to use a macro variable as the ENCRYPTKEY= key value. When you specify a macro variable for the ENCRYPTKEY= key value, you must enclose the macro variable in double quotation marks. If you do not use the double quotation marks, unpredictable results can occur. The following example defines a macro variable and uses the macro variable as the ENCRYPTKEY= key value:

```
%let secret=myvalue;
create table myschema.dsname {options encrypt=aes encryptkey="&secret"};
```

The following example uses the COPY statement from the DATASETS procedure and the SELECT statement:

```
copy in=OldLib out=NewLib;
select salary(encryptkey=key-value);
run;
```

The option can be specified either in parentheses after the name of the table or after a forward slash.

CAUTION

When using referential integrity constraints, all primary key and foreign key tables that reference each other must use the same encryption key. For more information, see the topic on encryption in *SAS Programmer's Guide: Essentials*. For more information about integrity constraints, see *SAS V9 LIBNAME Engine: Reference*.

You cannot change the key value on an AES-encrypted data file without re-creating the data file. The AUTHLIB procedure must be used to change the recorded key of a metadata-bound library.

Example: Using the ENCRYPTKEY= Option

This example uses the ENCRYPT=AES option to encrypt a SAS data set:

```
create table myfiles.salary {options encrypt=aes encryptkey="1234*#mykey"}
   (name char(15),
    yrsal double,
    bonuspct double);
insert into myfiles.salary {option encryptkey="1234*#mykey"} values
    ('Muriel', 34567, 3.2);
insert into myfiles.salary {option encryptkey="1234*#mykey"} values
    ('Bjorn', 74644, 2.5);
```

```
insert into myfiles.salary {option encryptkey="1234*#mykey"} values
    ('Freda', 38755, 4.1);
insert into myfiles.salary {option encryptkey="1234*#mykey"} values
    ('Benny', 29855, 3.5);
insert into myfiles.salary {option encryptkey="1234*#mykey"} values
    ('Agnetha', 70998, 4.1);
```

To retrieve data from the data set , you must specify the ENCRYPTKEY= option and the password:

```
  select * from myfiles.salary {option encryptkey="1234*#mykey"};
```

See Also

Table Options:

- "ENCRYPT= Table Option" on page 1039

ENDOBS= Table Option

Specifies the end observation number in a user-defined range of observations to be processed.

Valid in:	SELECT statement
Category:	Observation Control
Restrictions:	This table option is not supported on the CAS server. Use with input data sets only
Interaction:	Use in conjunction with STARTOBS= table option
Data source:	SPD Engine data set, SPD Server table

Syntax

ENDOBS= *n*

Arguments

n
is the number of the end observation.

Details

The software processes all of the observations in the entire data set unless you specify a range of observations with the STARTOBS= or ENDOBS= options. If the STARTOBS= option is used without the ENDOBS= option, the implied value of

ENDOBS= is the end of the data set. When both options are used together, the value of ENDOBS= must be greater than the value of STARTOBS=.

In contrast to the Base SAS software options FIRSTOBS= and OBS=, the STARTOBS= and ENDOBS= SPD Server options can be used for WHERE clause processing in addition to table input operations. When ENDOBS= is used in a WHERE expression, the ENDOBS= value represents the last observation to process, rather than the number of observations to return.

Example

The following code shows how the ENDOBS= table option is specified in the FedSQL SELECT statement:

```
select * from spdslib.table1(startobs=3 endobs=4);
```

See Also

Table Options:

- "STARTOBS= Table Option" on page 1069

EXTENDOBSCOUNTER= Table Option

Specifies whether to extend the maximum observation count in a new output SAS data set.

Valid in:	CREATE TABLE statement
Category:	Table Control
Alias:	EOC=
Default:	YES
Restriction:	This table option is not supported on the CAS server.
Data source:	SAS data set

Syntax

EXTENDOBSCOUNTER=YES | NO

Arguments

YES
requests an enhanced file format in a newly created SAS data set that counts observations beyond the 32-bit limitation. Although this SAS data set is created for an operating environment that stores the number of observations with a 32-bit

integer, the data set behaves like a 64-bit file with respect to counters. This is the default value.

Restrictions EXTENDOBSCOUNTER=YES is valid only for an output SAS data set whose internal data representation stores the observation count as a 32-bit integer. EXTENDOBSCOUNTER= is ignored for SAS data sets with a 64-bit integer.

A SAS data set that is created with an extended observation count is incompatible with releases prior to SAS 9.3.

NO
specifies that the maximum observation count in a newly created SAS data file is determined by the long integer size for the operating environment. In operating environments with a 32-bit integer, the maximum number is 2**31-1 or approximately two billion observations (2,147,483,647). In operating environments with a 64-bit integer, the maximum number is 2**63-1 or approximately 9.2 quintillion observations.

Details

Historically, Base SAS had a limitation in which up to approximately two billion observations could be counted and fully supported for operating environments with a 32-bit long integer. The EXTENDOBSCOUNTER= table option extends the limit to match that of operating environments with a 64-bit long integer. EXTENDOBSCOUNTER=NO is provided for backward compatibility.

GP_DISTRIBUTED_BY= Table Option

Specifies the distribution key for the table being created.

Category:	Table Control
Alias:	DISTRIBUTED_BY=
Restriction:	This table option is not supported on the CAS server.
Data source:	Greenplum, HAWQ

Syntax

GP_DISTRIBUTED_BY= 'DISTRIBUTED BY (*column*[, …*column*])
| DISTRIBUTED RANDOMLY'

Arguments

DISTRIBUTED BY (*column*, […*column*])
specifies one or more DBMS column names to use as the distribution key.

DISTRIBUTED RANDOMLY
specifies to determine the column or set of columns to use to distribute table rows across database segments. This is known as round-robin distribution.

Details

DISTRIBUTED BY uses hash distribution with one or more columns declared as the distribution key. For the most even data distribution, the distribution key should be the primary key of the table or a unique column (or set of columns). If that is not possible, then you might choose DISTRIBUTED RANDOMLY, which sends the data round-robin to the segment instances. If a value is not supplied, then hash distribution is chosen using the primary key (if the table has one) or the first eligible column of the table as the distribution key.

DISTRIBUTED_BY can be submitted as shown above or within the DBCREATE_TABLE_OPTS= table option. Here is an example of how it is specified in DBCREATE_TABLE_OPTS=:

```
dbcreate_table_opts='distributed by ("b")'
```

See Also

Table Options:

- “DBCREATE_TABLE_OPTS= Table Option” on page 1035

IDXNAME= Table Option

Directs SAS to use a specific index to match the conditions of a WHERE clause.

Category:	User Control of SAS Index Usage
Restrictions:	This table option is not supported on the CAS server. Use with input data sets only Cannot be used with the IDXWHERE= table option
Data source:	SAS data set

Syntax

IDXNAME= *index-name*

Arguments

index-name

specifies the name (up to 32 characters) of a simple or composite index for the SAS data set. SAS does not attempt to determine whether the specified index is the best one or whether a sequential search might be more resource efficient.

Interaction The specification is not a permanent attribute of the data set and is valid only for the current use of the data set.

Details

By default, to satisfy the conditions of a WHERE clause for an indexed SAS data set, SAS identifies zero or more candidate indexes that could be used to optimize the WHERE clause. From the list of candidate indexes, SAS selects the one that it determines will provide the best performance, or rejects all of the indexes if a sequential pass of the data is expected to be more efficient.

Because the index that SAS selects might not always provide the best optimization, you can direct SAS to use one of the candidate indexes by specifying the IDXNAME= table option. If you specify an index that SAS does not identify as a candidate index, then IDXNAME= table option does not process the request. That is, IDXNAME= does not allow you to specify an index that would produce incorrect results.

Comparisons

The IDXWHERE= table option enables you to override the SAS decision about whether to use an index.

Example

This example uses the IDXNAME= table option to direct SAS to use a specific index to optimize the WHERE clause. SAS then disregards the possibility that a sequential search of the data set might be more resource efficient and does not attempt to determine whether the specified index is the best one. (Note that the EMPNUM index was not created with the NOMISS option.)

```
create table mydata.empnew
   as select * from mydata.employee {option idxname=empnum}
   where empnum < 2000;
```

See Also

Table options

IDXWHERE= Table Option

Specifies whether SAS uses an index search or a sequential search to match the conditions of a WHERE clause.

Category: User Control of SAS Index Usage

Restrictions: This table option is not supported on the CAS server.
Use with input data sets only
SAS data sets: Cannot be used with IDXNAME=
SPD Engine data sets: IDXWHERE=NO cannot be used with WHERENOINDEX=

Data source: SAS data set, SPD Engine data set

Syntax

IDXWHERE= YES | NO

Arguments

YES
tells SAS to choose the best index to optimize a WHERE clause, and to disregard the possibility that a sequential search of the data set might be more resource-efficient. This is the default value.

NO
tells SAS to ignore all indexes and satisfy the conditions of a WHERE clause with a sequential search of the data set.

Notes You cannot use the IDXWHERE= table option to override the use of an index to process a BY statement.

You cannot use the WHERENOINDEX= table option when IDXWHERE=NO is used.

Details

By default, to satisfy the conditions of a WHERE clause for an indexed data set, the software decides whether to use an index or to read the data set sequentially. The software estimates the relative efficiency and chooses the method that is more efficient.

You might need to override the software's decision by specifying the IDXWHERE= table option because the decision is based on general rules that occasionally might not produce the best results. That is, by specifying the IDXWHERE= table option, you are able to determine the processing method.

Note: The specification is not a permanent attribute of the data set and is valid only for the current use of the data set.

Comparisons

The IDXNAME= table option (which is supported only for SAS data sets) enables you to direct SAS to use a specific index.

The WHERENOINDEX= table option enables you to specify a list of indexes to exclude when making WHERE expression evaluations.

Examples

Example 1: Specifying Index Usage

This example uses the IDXWHERE= table option to tell SAS to decide which index is the best to optimize the WHERE clause. SAS then disregards the possibility that a sequential search of the data set might be more resource-efficient:

```
create table mydata.empnew
   as select * from mydata.employee {option idxwhere=yes}
   where empnum < 2000;
```

Example 2: Specifying No Index Usage

This example uses the IDXWHERE= table option to tell SAS to ignore any index and to satisfy the conditions of the WHERE clause with a sequential search of the data set:

```
create table mydata.empnew
   as select * from mydata.employee {option idxwhere=no}
   where empnum < 2000;
```

See Also

Table options:

- “IDXNAME= Table Option” on page 1048
- “WHERENOINDEX= Table Option” on page 1099

IOBLOCKSIZE= Table Option

Specifies the number of rows in a block to be used in an I/O operation.

Category: Table Control

Restriction: This table option is not supported on the CAS server.

Data source: SPD Engine data set, SPD Server table

Syntax

IOBLOCKSIZE= *n*

Arguments

n

specifies the size of the block in bytes. The default value is smallest block size supported by the data source.

Details

The software reads and stores rows in the table in blocks. IOBLOCKSIZE= is useful on compressed or encrypted tables. The software does not use IOBLOCKSIZE= on noncompressed or nonencrypted tables.

For tables that you compress or encrypt, the IOBLOCKSIZE= specification determines the number of rows to include in the block. The specification applies to block compression as well as data I/O to and from disk. The IOBLOCKSIZE= value affects the table's organization on disk.

When using compression or encryption, specify an IOBLOCKSIZE= value that complements how the data is to be accessed, sequentially or randomly. Sequential access or operations requiring full table scans favor a large block size. In contrast, random access favors a smaller block size. On SPD Engine, a large block size would be 131,072 bytes. The smallest allowed value is 32,768 bytes. For SPD Server, a large block size is 64,000 bytes. The smallest allowed value is 8,000 bytes.

See Also

Table Options:

- “COMPRESS= Table Option” on page 1030
- “ENCRYPT= Table Option” on page 1039
- “PADCOMPRESS= Table Option” on page 1058

KMS_PATH= Table Option

Specifies the location of a key file resource within a project.

Category: Table Access

Alias: KMS_FILE=

Restriction: This table option is not supported on the CAS server.

Data source: Google BigQuery

Note: Support for this option was added in the November 2019 release of SAS/ACCESS.

Example: `kms_path="/u/BigQuery/projFiles/filename"`

Syntax

KMS_PATH="*path-and-filename*"

Arguments

"*path-and-filename*"
specifies the path to the key file resource within a project.

See Also

Table Options:

- "LOCATION= Table Option" on page 1054

LABEL= Table Option

Specifies a label for a table.

Category: Table Control

Restriction: This table option is not supported in the CAS server.

Data source: SAS data set, SPD Engine data set

Syntax

LABEL= '*label* '

Arguments

'label'

specifies a text string of up to 256 characters.

Requirement For SAS data sets and SPD Engine data sets, if the text contains single quotation marks, use double quotation marks around the label. Or, use two single quotation marks in the label text and enclose the string in single quotation marks. To remove a label from a data set, assign a label that is equal to a blank that is enclosed in quotation marks.

Details

The labels specified with the LABEL= table option are stored as part of the table's metadata; however, the information is not used by FedSQL. That is, once stored, the label cannot be displayed. The label can be viewed with the SAS CONTENTS procedure.

You can use the LABEL= table option on both input and output tables. When you use the LABEL= table option on input tables, it assigns a label for the file for the duration of the operation. When this is specified for an output table, the label becomes a permanent part of that file.

A label assigned to a table remains associated with that table when you update a table in place, such as when you use the APPEND procedure or the MODIFY statement. However, a label is lost if you use a table with a previously assigned label to create a new table with the CREATE TABLE statement. For example, a label previously assigned to table ONE is lost when you create the new output table ONE in this CREATE TABLE statement:

```
create table one as select * from one;
```

Example

These examples assign labels to SAS data sets:

```
create table tst {option label='1976 W2 Info, Hourly'} (col1 char(4),
col2 double);
create table tst1 {option label="Hillside's Daily Account"} as select
* from tst;
create table tst2 {option label='Peter''s List'}(col1 char(20), col2
char(20));
```

LOCATION= Table Option

Specifies the location of the data set.

Category: Table Access

Restriction: This table option is not supported in the CAS server.

Data source: Google BigQuery

Note: Support for this option was added in the November 2019 release of SAS/ACCESS.

Syntax

LOCATION='*value*'

Argument

'*value*'
specifies the location of the data set. For valid values, see https://cloud.google.com/bigquery/docs/locations.

Details

If LOCATION= is not specified, BigQuery determines the location.

Examples

Example 1

This setting specifies the location for the United States:

```
LOCATION='US'
```

Example 2

This setting specifies the location for the European Union:

```
LOCATION='EU'
```

See Also

Table Options:

- “KMS_PATH= Table Option” on page 1053

LOCKTABLE= Table Option

Places shared or exclusive locks on tables.

Category: Table Control

Restriction: This table option is not supported on the CAS server.

Data source: SAS data set

Syntax

LOCKTABLE= |

Arguments

SHARE
locks a table in shared mode, allowing other users or processes to read data from the tables, but preventing users from updating data.

EXCLUSIVE
locks a table exclusively, preventing other users from accessing any table that you open.

Details

You can lock tables only if you are the owner or have been granted the necessary privilege.

If you access the BASE table driver through PROC FEDSQL, the default value for the LOCKTABLE option is EXCLUSIVE. However, if you access the BASE table driver through a SAS Federation Server, or if you run your program locally with the SAS Federation Server LIBNAME engine, the default value for the LOCKTABLE option is SHARE.

ORHINTS= Table Option

Specifies Oracle hints to pass to Oracle from FedSQL.

Valid in: DELETE statement, INSERT statement, SELECT statement, UPDATE statement

Category: Data Control

Restriction: This table option is not supported on the CAS server.

Data source: Oracle

Syntax

ORHINTS='/* *Oracle-hint* */'

Arguments

Oracle-hint
specifies an Oracle hint for the FedSQL language processor to pass to the DBMS as part of a query. If you do not specify a hint, FedSQL does not send any hints to Oracle. For more information, see *SAS Federation Server: Administrator's Guide*.

Details

The ORHINTS= table option is used in conjunction with the DRIVER_TRACE= and DRIVER_TRACEFILE= data source connection options. PROC FEDSQL and PROC DS2 do not support use of data source connection options.

Example: Using the ORHINTS= Option

These examples show how to specify ORHINTS=:

```
select * from test{options orhints='/* dummy hint */'} ;

update test{options orhints='/* dummy hint */'}  set c1=1;

insert into test{options orhints='/* dummy hint */'}  values (100);

delete from test{options orhints='/* dummy hint */'} ;
```

ORNUMERIC= Table Option

Specifies how numbers read from or inserted into the Oracle NUMBER column are treated.

Category:	Table Control
Default:	YES
Restriction:	This table option is not supported on the CAS server.
Data source:	Oracle
Data type:	DECIMAL, NUMERIC

Syntax

ORNUMERIC=YES | NO

Arguments

NO
Indicates that the numbers are treated as TKTS_DOUBLE values. They might not have precision beyond 14 digits.

YES
Indicates that non-integer values with explicit precision are treated as TKTS_NUMERIC values. This is the default setting.

Details

This option defaults to YES so that a NUMBER column with precision or scale is described as TKTS_NUMERIC. This option can be specified as both a connection option and a table option. When specified as both connection and table option, the table option value overrides the connection option.

PADCOMPRESS= Table Option

Specifies the number of bytes to add to compressed blocks in an SPD Engine data set that is opened for OUTPUT or UPDATE.

Category:	Table Control
Restriction:	This table option is not supported on the CAS server.
Data source:	SPD Engine data set

Syntax

PADCOMPRESS= *n*

Arguments

n
specifies the number of bytes to add. The default number is 0 (zero).

Details

Compressed SPD Engine data sets occupy blocks of space on the disk. The size of a block is derived from the IOBLOCKSIZE= table option that is specified when the data set is created. When the data set is updated, a new block fragment might need to be created to hold the update. More updates might then create new fragments, which, in turn, increases the number of I/O operations needed to read a data set.

By increasing the block padding in certain situations where many updates to the data set are expected, fragmentation can be kept to a minimum. However, adding padding can waste space if you do not update the data set.

You must weigh the cost of padding all compression blocks against the cost of possible fragmentation of some compression blocks.

Specifying the PADCOMPRESS= table option when you create or update a data set adds space to all of the blocks as they are written back to the disk. The PADCOMPRESS= setting is not retained in the data set's metadata.

See Also

Table Options:

- "COMPRESS= Table Option" on page 1030
- "IOBLOCKSIZE= Table Option" on page 1051

PARTSIZE= Table Option

Specifies the size of the table partitions.

Valid in:	CREATE TABLE statement
Category:	Table Control
Restriction:	This table option is not supported on the CAS server.
Data source:	SPD Engine data set, SPD Server table

Syntax

PARTSIZE= *n*

Arguments

n

specifies the size of the partition.

Requirements The size can be specified in megabytes, gigabytes, and terabytes. If *n* is specified without M, G, or T, the default is

megabytes. For example, PARTSIZE=128 is the same as PARTSIZE=128M. For SPD Engine data sets, the default value is 128 megabytes. The maximum value is 8,796,093,022,207 megabytes. For SPD Server tables, the default value is the setting of the MINPARTSIZE= server option. If the MINPARTSIZE= server option is not set, the default is 16 megabytes.

When creating an SPD Engine data set, if the row length is greater than 65K, you might find that the PARTSIZE= that you specify and the actual partition size do not match. To get these numbers to match, specify a PARTSIZE= that is a multiple of 32 and the row length.

Details

For more information, see “PARTSIZE= Data Set Option in *SAS Scalable Performance Data Engine: Reference* and “PARTSIZE= Table Option” in *SAS Scalable Performance Data Server: User’s Guide*, as appropriate.

PARTITION_KEY= Table Option

Specifies the column name to use as the partition key for creating fact tables.

Category: Table Control

Restriction: This table option is not supported on the CAS server.

Data source: Aster

Syntax

PARTITION_KEY= "*column-name*"

Arguments

column-name
specifies the name of the column, in quotation marks.

Details

Aster uses two table types: dimension and fact. To create a fact table in Aster without error, you must set the PARTITION_KEY= table option.

PASSWORD= Table Option

Specifies the password for the HDFS user.

Category: Bulk Loading

Alias: PASS=, PWD=

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028.

Interaction: Used in conjunction with "USER= Table Option" on page 1098.

Data source: Impala

Syntax

PASSWORD="*value*"

Arguments

"*value*"
specifies the password for the HDFS user.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026
- "USER= Table Option" on page 1098

PICKLIST= Table Option

Specifies the picklist to use for the bulk-loading operation.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028.

Data source: Impala

Syntax

PICKLIST="*filename*"

Arguments

"*filename*"
specifies the full pathname to a text file that contains the Hadoop picklist.

Default When this table option is omitted, the file “hadoopbasics/hadoopwrapr.txt” is used.

Details

Use the PICKLIST= option when you want to override the default picklist of “hadoopbasics/hadoopwrapr.txt”.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 1026

POINTOBS= Table Option

Specifies whether SAS creates compressed data sets whose observations can be randomly accessed or sequentially accessed.

Category:	Table Control
Restrictions:	This table option is not supported on the CAS server. Use with output tables only.
Interaction:	Used with COMPRESS= table option
Data source:	SAS data set

Syntax

POINTOBS= YES | NO

Syntax Description

YES

causes the FedSQL language to produce a compressed data set that might be randomly accessed by observation number. This is the default.

Here are examples of accessing data directly by observation number:

- the POINT= option of the MODIFY and SET statements in the DATA step
- going directly to a specific observation number with PROC FSEDIT.

> **TIP** Specifying POINTOBS=YES does not affect the efficiency of retrieving information from a data set. It does increase CPU usage by approximately 10% when creating a compressed data set and when updating or adding information to it.

NO

suppresses the ability to randomly access observations in a compressed data set by observation number.

> **TIP** Specifying POINTOBS=NO is desirable for applications where the ability to point directly to an observation by number within a compressed data set is not important. If you do not need to access data by observation number, then you can improve performance by approximately 10% by specifying POINTOBS=NO:
>
> - when creating a compressed data set
> - when updating or adding observations to it

Details

Note that REUSE=YES takes precedence over POINTOBS=YES. For example:

```
create table test{options compress=yes pointobs=yes reuse=yes};
```

This example code results in a data set that has POINTOBS=NO. Because POINTOBS=YES is the default when you use compression, REUSE=YES causes POINTOBS= to change to NO.

See Also

Table Options:

- “COMPRESS= Table Option” on page 1030
- “REUSE= Table Option” on page 1068

POST_TABLE_OPTS= Table Option

Allows database-specific options to be placed after the table name in the CREATE TABLE statement.

Category:	Table Control
Default:	None
Restriction:	This table option is not supported on the CAS server.
Data source:	Hive, Spark
Note:	This option is available beginning with SAS 9.4M5.
See:	“DBCREATE_TABLE_OPTS= Table Option” on page 1035 “PRE_TABLE_OPTS= Table Option” on page 1065

Syntax

POST_TABLE_OPTS="*DBMS-SQL-options*"

Required Argument

"*DBMS-SQL-options*"
specifies additional database-specific options to be placed after the table name in a CREATE TABLE statement. You can use single or double quotation marks around the DBMS value.

Details

You can use the POST_TABLE_OPTS= table option alone or with these related table options: PRE_TABLE_OPTS= and POST_STMT_OPTS=. POST_STMT_OPTS= is an alias for DBCREATE_TABLE_OPTS=. For example, you can supply database options according to these templates:

```
create table dblib.test{OPTIONS post_table_opts="external"} col1(int);

create table dblib.test{options pre_table_opts="external"
                                post_table_opts="stored as orc"} col1(int);


create table dblib.test{options pre_table_opts="external"
                                post_table_opts="stored as"
                                post_stmt_options="orc"} col1(int);
```

When all three table options are specified together, they are processed as follows:

```
CREATE [PRE_TABLE_OPTIONS] TABLE test (col1 int) [POST_TABLE_OPTS]
[POST_STMT_OPTS/DBCREATE_TABLE_OPTS]
```

PRE_TABLE_OPTS= Table Option

Allows database-specific options to be placed before the table name in the CREATE TABLE statement.

Category:	Table Control
Default:	None
Restriction:	This table option is not supported in the CAS server.
Data source:	Hive, Spark
Note:	This option is available beginning with SAS 9.4M5.
See:	"DBCREATE_TABLE_OPTS= Table Option" on page 1035 "POST_TABLE_OPTS= Table Option" on page 1064

Syntax

PRE_TABLE_OPTS="*DBMS-SQL-options*"

Required Argument

"*DBMS-SQL-options*"
specifies additional database-specific options to be placed before the table name in a CREATE TABLE statement. You can use single or double quotation marks around the DBMS value.

Details

You can use the PRE_TABLE_OPTS= table option alone or with these related table options: POST_TABLE_OPTS= and POST_STMT_OPTS=. POST_STMT_OPTS= is an alias for DBCREATE_TABLE_OPTS=. For example, you can supply database options according to these templates:

```
create table dblib.test{OPTIONS pre_table_opts="external"} col1(int);

create table dblib.test{OPTIONS pre_table_opts="external"
                                post_table_opts="stored as orc"} col1(int);


create table dblib.test{OPTIONS pre_table_opts="external"
                                post_table_opts="stored as"
                                post_stmt_opts="orc"} col1(int);
```

When all three table options are specified together, they are processed as follows:

```
CREATE [PRE_TABLE_OPTIONS] TABLE test (col1 int) [POST_TABLE_OPTS]
[POST_STMT_OPTS/DBCREATE_TABLE_OPTS]
```

PW= Table Option

Assigns a READ, WRITE, and ALTER password to a SAS data set or an SPD Engine data set and enables access to the password-protected file. Specifies a key value for accessing an encrypted SPD Server table.

Category: Table Control

Restriction: This table option is not supported on the CAS server.

Data source: SAS data set, SPD Engine data set, SPD Server table

Note: Check your log after this operation to ensure that the password values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in *SAS Programmer’s Guide: Essentials*.

Syntax

PW= *password* | *key-value*

Arguments

password
must be a valid SAS name.

key-value
must be a valid SAS name.

Details

The PW= table option applies to a SAS data set, an SPD Engine data set, and an SPD Server table. For a SAS data set and SPD Engine data set, you can use this option to assign a password to the data set or to access a password-protected data set. When a data set that is protected by a password is replaced, the new data set inherits the password.

Note: A SAS password does not control access to a SAS file beyond the SAS system. You should use the operating system-supplied utilities and file-system security controls in order to control access to SAS files outside of SAS.

For SPD Server tables, you can use the PW= option to access an SPD Server table that is already encrypted. The PW= option enables access to table files that are protected by the SAS proprietary encryption algorithm.

Examples

Example 1: Assigning a Password or Encryption Key with PW=

```
create table myfiles.mytable {options pw=green}(column1 double);
```

Example 2: Specifying a Password or Encryption Key with PW=

```
select * from myfiles.mytable {option pw=green};
```

READ= Table Option Table Option

Assigns a READ password to a SAS file that prevents users from reading the file, unless they enter the password.

Category:	Table Control
Restriction:	This table option is not supported on the CAS server.
Data source:	SAS data set, SPD Engine data set
Note:	Check your log after this operation to ensure that the password values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in *SAS Programmer’s Guide: Essentials*.

Syntax

READ= *read-password*

Arguments

read-password
must be a valid SAS name.

Details

The READ= option applies only to a SAS data set or an SPD Engine data set. You can use this option to assign a password or to access a Read-protected file.

Note: A SAS password does not control access to a SAS file beyond SAS. You should use the operating system-supplied utilities and file-system security controls in order to control access to SAS files outside SAS.

REUSE= Table Option

Specifies whether new rows can be written to freed space in a compressed SAS data set.

Category: Table Control

Restrictions: This table option is not supported on the CAS server.
Use with output data sets only

Data source: SAS data set

Syntax

REUSE= NO | YES

Arguments

NO
does not track and reuse space in a compressed SAS data set. New rows are appended to the existing data set. Specifying NO results in less efficient data storage if you delete or update many rows in the data set.

YES
tracks and reuses space in a compressed SAS data set. New rows are inserted in the space that is freed when other rows are updated or deleted.

If you plan to use operations that add rows to the end of a compressed data set, use REUSE=NO. REUSE=YES causes new rows to be added wherever there is space in the file, not necessarily at the end of the file.

Details

By default, new rows are appended to an existing compressed data set. If you want to track and reuse free space by deleting or updating other rows, use the REUSE= table option when you create a compressed SAS data set.

The REUSE= table option has meaning only when you are creating a new data set with the COMPRESS=YES option. Using the REUSE= table option when you are accessing an existing SAS data set has no effect.

See Also

Table Options:

- "COMPRESS= Table Option" on page 1030
- "POINTOBS= Table Option" on page 1062

STARTOBS= Table Option

Specifies the starting row number in a user-defined range of rows to be processed.

Valid in:	SELECT statement
Category:	Observation Control
Restrictions:	This table option is not supported on the CAS server. Use with input data sets only
Data source:	SPD Engine data set, SPD Server tables

Syntax

STARTOBS= *n*

Arguments

n
is the number of the starting row.

Details

The software processes the entire data set unless you specify a range of rows with the STARTOBS= and ENDOBS= options. If the ENDOBS= option is used without the STARTOBS= option, the implied value of STARTOBS= is 1. When both options are used together, the value of STARTOBS= must be less than the value of ENDOBS=.

In contrast to the Base SAS software options FIRSTOBS= and OBS=, the STARTOBS= and ENDOBS= options can be used for WHERE clause processing in addition to table input operations. When STARTOBS= is used in a WHERE expression, the STARTOBS= value represents the first row on which to apply the WHERE expression.

Example

The following code shows how the STARTOBS= table option is specified in the FedSQL SELECT statement:

```
select * from spdslib.table1(startobs=3 endobs=4);
```

See Also

Table Options:

- "ENDOBS= Table Option" on page 1045

TABLE_TYPE= Table Option

Specifies the type of table storage FedSQL will use when creating tables in SAP HANA.

Valid in:	CREATE TABLE statement
Category:	Data Access
Restriction:	This table option is not supported on the CAS server.
Interactions:	GLOBAL and GLOBAL TEMPORARY have the same behavior LOCAL and LOCAL TEMPORARY have the same behavior
Data source:	SAP HANA

Syntax

TABLE_TYPE= [COLUMN] [GLOBAL] [GLOBAL TEMPORARY] [LOCAL] [LOCAL TEMPORARY] [ROW]

Arguments

COLUMN
creates a table using column-based storage in SAP HANA.

GLOBAL | GLOBAL TEMPORARY
creates a global, temporary table in SAP HANA. The tables are globally available; however, the data is visible only in the current session.

LOCAL | LOCAL TEMPORARY
creates a local, temporary table in SAP HANA. The table definition and data are visible only in the current session.

ROW
creates a table using row-based storage in SAP HANA.

Details

The SAP HANA TABLE_TYPE= option is available as a connection option and as a table option. If neither are specified, tables that are created in SAP HANA follow the SAP HANA default for row or column store.

See Also

Table Options:

- "DBCREATE_TABLE_OPTS= Table Option" on page 1035

TD_BUFFER_MODE= Table Option

Specifies whether the LOAD method is used.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	Teradata

Syntax

TD_BUFFER_MODE= YES | NO

Arguments

YES
enables the bulk load feature. This option must be set to YES for the bulk load feature to work.

NO
disables the bulk load feature. This is the default value.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

TD_CHECKPOINT= Table Option

Specifies when the TPT operation issues a checkpoint or savepoint to the database.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028

Data source: Teradata

Syntax

TD_CHECKPOINT= *number*

Arguments

number
specifies the number of rows after which the TPT operation issues a checkpoint or savepoint to the database. The default is 0, which means no checkpoint is taken. All rows are saved at the end of the job.

TD_DATA_ENCRYPTION= Table Option

Activates data encryption.

Category: Bulk Loading

Alias: TPT_DATA_ENCRYPTION=

Default: NO

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028

Data source: Teradata

Syntax

TD_DATA_ENCRYPTION= YES | NO

Arguments

YES
encrypts all SQL requests, responses, and data.

Alias ON

NO
no encryption occurs. This is the default setting.

Alias OFF

Details

The TD_DATA_ENCRYPTION= session attribute controls encryption for all connections that the Teradata TPT (Teradata Parallel Transporter) creates for LOAD, UPDATE, and STREAM operations.

The TD_DATA_ENCRYPTION= table option overrides the setting of the TPT_DATA_ENCRYPTION= LIBNAME option.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

TD_DROP_ERROR_TABLE= Table Option

Drops the log table at the end of the job, whether the job was completed successfully or not.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	Teradata

Syntax

TD_DROP_ERROR_TABLE= YES | NO

Arguments

YES
specifies to drop the error tables at the end of the job, whether the job was completed successfully or not.

NO
keeps error tables that contain errors after the job is complete. Error tables that are empty after successful completion are deleted. This is the default setting.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

TD_DROP_LOG_TABLE= Table Option

Drops the log table at the end of the job, whether the job completed successfully or not.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	Teradata

Syntax

TD_DROP_LOG_TABLE= YES | NO

Arguments

YES
specifies to drop the log table at the end of the job, whether the job is completed successfully or not.

NO
keeps log tables for jobs that were not completed successfully. This is the default setting.

TD_DROP_WORK_TABLE= Table Option

Drops the work table at the end of the job, whether the job was completed successfully or not.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028

Data source: Teradata

Syntax

TD_DROP_WORK_TABLE= YES | NO

Arguments

YES
specifies to drop the work table at the end of the job, whether the job was completed successfully or not.

NO
keeps work tables for jobs that were not completed successfully. This is the default setting.

TD_ERROR_LIMIT= Table Option

Specifies the maximum number of records that can be stored in an error table.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028

Data source: Teradata

Syntax

TD_ERROR_LIMIT= *number-of-records*

Arguments

number-of-records
specifies the maximum number of records that can be stored in an error table before the Load, Stream, or Update operator job is terminated. By default, the ErrorLimit value is unlimited. The number of records must be greater than zero.

Details

The ErrorLimit specification applies to each instance of an operator job. Specifying an invalid value terminates the job.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

TD_ERROR_TABLE_1= Table Option

Specifies a name for the first error table.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKLOAD= Table Option" on page 1026
Data source:	Teradata

Syntax

TD_ERROR_TABLE_1= *name-of-table*

Arguments

name-of-table

specifies a name for the error table. The default name for ErrorTable1 is *ttname*_ET, where *ttname* is the name of the target table. Table names exceeding 27 characters are truncated to accommodate the three-character suffix. Therefore, you might want to specify a name for the table that will not be truncated. When truncation occurs, a message is written to the log.

Restriction The name of an existing table cannot be used, unless you are restarting a paused job.

Details

ErrorTable1 contains records that were rejected during the acquisition phase of a Load, Stream, or Update operator job because of the following:

- Data conversion errors

- Constraint violations
- Access Module Processor (AMP) configuration changes.

The error table is created in the default user (logon) database, optionally qualified with a schema, unless the TD_LOGDB= table option is used to specify a different location for utility tables.

Error tables that contain errors are retained at the end of a job. Specify the TD_DROP_ERROR_TABLE= table option if you do not want to retain them.

For information about the error table format and the procedure to correct errors, see *Teradata Parallel Transporter Reference*.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026
- "TD_DROP_ERROR_TABLE= Table Option" on page 1073
- "TD_LOGDB= Table Option" on page 1081

TD_ERROR_TABLE_2= Table Option

Specifies a name for the second error table.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	Teradata

Syntax

TD_ERROR_TABLE_2= *name-of-table*

Arguments

name-of-table

specifies a name for the error table. The default name for ErrorTable2 is *ttname*_UV, where *ttname* is the name of the target table. Table names exceeding 27 characters are truncated to accommodate the three-character suffix. Therefore, you might want to specify a name for the table that will not be truncated. When truncation occurs, a message is written to the log.

Restrictions This option is provided for the Load and Update operators. It is ignored by the Stream operator.

The name of an existing table cannot be used, unless you are restarting a paused job.

Details

ErrorTable2 contains records that violated the unique primary index constraint. This type of error occurs during the application phase of a Load or Update operator job.

The error table is created in the default user (logon) database, optionally qualified with a schema, unless the TD_LOGDB= table option is used to specify a different location for utility tables.

For information about the error table format and the procedure to correct errors, see *Teradata Parallel Transporter Reference*.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026
- "TD_LOGDB= Table Option" on page 1081

TD_LOG_MECH_TYPE= Table Option

Specifies the logon mechanism for a bulk load.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	Teradata

Syntax

TD_LOG_MECH_TYPE= *mechanism*

Arguments

mechanism

specifies the logon authentication mechanism. Currently, the valid value is LDAP.

Requirement The driver requires an 8-character input value. To submit, pad the value with four spaces as follows:

```
TD_LOG_MECH_TYPE="LDAP    "
```

Details

If the user connected via LDAP with the UID connection option, there is no reason to use this option. Use TD_LOG_MECH_TYPE= and TD_LOG_MECH_DATA= to allow LDAP authentication for TPT if the user did not specify LDAP authentication for UID in the connect string. You can also use this option with TD_LOG_MECH_DATA= to override the LDAP ID specified in the UID option with a different LDAP ID.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 1026

TD_LOG_MECH_DATA= Table Option

Specifies additional data for the logon mechanism.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the “BULKOPTS= Table Option” on page 1028
Data source:	Teradata

Syntax

TD_LOG_MECH_DATA="*string***"**

Arguments

string

used in conjunction with the TD_LOG_MECH_TYPE= table option, specifies credentials for authentication. Currently, only LDAP credentials are accepted. The credentials must be in the following form:

```
TD_LOG_MECH_DATA="authcid=value  password=value  realm=value"
```

See Also

Table Options:

- “BULKLOAD= Table Option” on page 1026
- “TD_LOG_MECH_TYPE= Table Option” on page 1078

TD_LOG_TABLE= Table Option

Specifies the name of the restart log table.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the “BULKOPTS= Table Option” on page 1028
Data source:	Teradata

Syntax

TD_LOG_TABLE= *name-of-table*

Arguments

name-of-table
specifies the name of the restart log table for the Load, Stream, and Update operators. The restart log table contains restart information. The default name for the restart log table is *ttname*_RS, where *ttname* is the name of the target table. Table names exceeding 30 characters are truncated to accommodate the three-character suffix. Therefore, you might want to specify a name for the table that will not be truncated. When truncation occurs, a message is written to the log.

Details

The restart log table is created in the user's default (logon) database, unless the TD_LOGDB= table option is used to specify a different location.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 1026
- “TD_DROP_LOG_TABLE= Table Option” on page 1074

- "TD_LOGDB= Table Option" on page 1081
- "TD_WORKING_DB= Table Option" on page 1095

TD_LOGDB= Table Option

Specifies the database where the TPT utility tables are created.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028

Data source: Teradata

Syntax

TD_LOGDB= *database-name*

Arguments

database-name
specifies the name of the database where the utility tables are to be created. If this option is not specified, the utility tables are created in the specified SCHEMA. If SCHEMA is not specified, the tables are created in the default user (logon) database. The utility tables include ErrorTable1, ErrorTable2, the restart log table, and the work table.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

TD_MAX_SESSIONS= Table Option

Specifies the maximum number of logon sessions that TPT can acquire for a job.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028

Data source: Teradata

Syntax

TD_MAX_SESSIONS= *integer*

Arguments

integer
specifies the maximum number of logon sessions that the Teradata Parallel Transporter can acquire for an operator job. The default value is four sessions, if a value is not specified. The maximum value cannot be more than the number of AMPs available. See your Teradata documentation for details.

Details

The TD_MAX_SESSIONS= value must be greater than zero. Specifying a value less than 1 causes the job to terminate.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

TD_MIN_SESSIONS= Table Option

Specifies the minimum number of sessions for TPT to acquire before a job starts.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	Teradata

Syntax

TD_MIN_SESSIONS= *integer*

Arguments

integer
specifies the minimum number of sessions required for TPT to start a job. The number is applied to the Load, Stream, and Update operators. The default is one session.

Details

The TD_MIN_SESSIONS= value must be greater than zero and less than or equal to the maximum number of sessions. Specifying a value less than 1 causes the active operator to terminate.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

TD_NOTIFY_LEVEL= Table Option

Specifies the level at which log events are recorded.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified with the "BULKOPTS= Table Option" on page 1028
Data source:	Teradata

Syntax

TD_NOTIFY_LEVEL= *level*

Arguments

level
Specifies the level at which events are reported. Valid settings are any of the following:

OFF	No notification of events is provided (the default value).
LOW	Initialize, CLIv2/DBS error, Exit events only
MEDIUM	All events except Err1 and Err2

HIGH All events.

Details

If you set TD_NOTIFY_LEVEL=, set TD_NOTIFY_METHOD= to indicate how you want the events reported.

See Also

Table Options:

- "TD_NOTIFY_METHOD= Table Option" on page 1084

TD_NOTIFY_METHOD= Table Option

Specifies the method for reporting events.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified with the "BULKOPTS= Table Option" on page 1028
Data source:	Teradata

Syntax

TD_NOTIFY_METHOD= *method*

Arguments

method

specifies the notify method to be used for reporting events. Valid settings are any of the following:

NONE	No notification of events is provided (the default value)
MSG	Sends events to a log. On Windows, the events are sent to an EventLog that can be viewed using the Event Viewer. On UNIX, the destination of events is specified in the /etc/syslog.conf file.
EXIT	Sends events to a user-defined notify exit routine.

Details

If you set TD_NOTIFY_METHOD=, set TD_NOTIFY_LEVEL= to indicate the types of events that you want reported.

See Also

Table Options:

- "TD_NOTIFY_LEVEL= Table Option" on page 1083
- "TD_NOTIFY_STRING= Table Option" on page 1085

TD_NOTIFY_STRING= Table Option

Defines a string that precedes all messages sent to the system log.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified with the "BULKOPTS= Table Option" on page 1028
Data source:	Teradata

Syntax

TD_NOTIFY_STRING= *string*

Arguments

string

Is an optional, user-specified string that precedes all messages that are sent to the system log. This string is also sent to the user-defined notify exit routine. If the NotifyMethod is MSG, the maximum length is 16 bytes. If the NotifyMethod is EXIT, the maximum length is 80 bytes.

See Also

Table Options:

- "TD_NOTIFY_METHOD= Table Option" on page 1084

TD_PACK= Table Option

Specifies the number of statements to pack into a multistatement request.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028

Data source: Teradata

Syntax

TD_PACK= *number*

Arguments

number

specifies the number of statements to pack into a multistatement request. The default number is 20. The maximum number of statements allowed is 600.

Restriction This option is ignored by the Load and Update operators.

See Also

Table Options:

- "TD_PACK_MAXIMUM= Table Option" on page 1086

TD_PACK_MAXIMUM= Table Option

Lets the Stream operator determine the pack factor for the current Stream job.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028

Data source: Teradata

Syntax

TD_PACK_MAXIMUM= YES | NO

Arguments

YES
triggers the Stream operator to dynamically determine the maximum pack factor for the current Stream job.

Restriction This option is ignored by the Load and Update operators.

NO
loads the number of statements indicated by the TD_PACK= option for multistatement requests.

See Also

Table Options:

- "TD_PACK= Table Option" on page 1086

TD_PAUSE_ACQ= Table Option

Forces a pause between the acquisition phase and the application phase of a load job.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified with the "BULKOPTS= Table Option" on page 1028
Data source:	Teradata

Syntax

TD_PAUSE_ACQ= YES | NO

Arguments

YES
specifies to pause the load job after the acquisition phase and skip the application phase.

Restriction This option is provided for the Load and Update operators. It is ignored by the Stream operator.

NO
distributes all of the rows that were sent to the Teradata database during the acquisition phase to their final destination on the AMPs. This is the default value.

TD_SESSION_QUERY_BAND= Table Option

Passes a string of user-specified name=value pairs for use by the TPT session.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified with the "BULKOPTS= Table Option" on page 1028

Data source: Teradata

Syntax

TD_SESSION_QUERY_BAND= *string*

Arguments

string
specifies one or more query band expressions for use by each TPT session generated for the job or process. The bands are specified as name=value pairs.

Details

The bands are passed to the database for use by the server for load balancing. They are also used as trigger mechanisms by Teradata Multi-System Manager or TASM (Teradata Active System Management).

TD_TENACITY_HOURS= Table Option

Specifies the amount of time the TPT operator continues trying to log on to the Teradata database.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028

Data source: Teradata

Syntax

TD_TENACITY_HOURS= *integer*

Arguments

integer
specifies the number of hours the TPT operator should attempt to log on to the Teradata database, when the maximum number of server utility slots are already occupied by other jobs. The default value is 0 (zero), meaning the operator does not try again and the job fails. It is recommended that you set a value when the Load and Update operators are used.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

TD_TENACITY_SLEEP= Table Option

Specifies the amount of time the TPT operator pauses, before retrying to log on to the Teradata database.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	Teradata

Syntax

TD_TENACITY_SLEEP= *integer*

Arguments

integer
specifies the number of minutes that the TPT operator pauses before retrying a job. This option is activated only when TD_TENACITY_HOURS= is set to a value. The default is six minutes.

Details

The TD_TENACITY_SLEEP= value must be greater than zero. If you specify a value of less than 1, the TPT operator responds with an error message and terminates the job.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

TD_TPT_OPER= Table Option

Specifies the load operator that is used by the Teradata Parallel Transporter.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified with the "BULKOPTS= Table Option" on page 1028
Data source:	Teradata

Syntax

TD_TPT_OPER= *operator*

Arguments

operator

Specifies the load operator that the Teradata Parallel Transporter (TPT) uses to load data. Valid values are LOAD, STREAM, or UPDATE.

LOAD	specifies the Load operator, which inserts into an empty table. This is the fastest of the load operators.
STREAM	specifies the Stream operator. It can be used to insert rows into empty Teradata tables or append to existing tables. The Stream operator is used by default when the TD_TPT_OPER= table option is not specified.
UPDATE	specifies the Update operator, which inserts into existing tables. The Update operator is faster than the Stream operator.

Details

When BULKLOAD=YES is specified, the Teradata Parallel Transporter (TPT) is used to load data. TPT jobs are run using operators. These are discrete object-oriented modules that perform specific extraction, loading, and updating processes.

To obtain the best performance when the Update operator is used, it is recommended that you drop all unique secondary indexes, foreign key references, and join indexes onto target tables before the load.

The Teradata server supports a maximum of 16 concurrent utility slots. The Stream operator does not take up a utility slot. The Load and Update operators do take up a slot.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026
- "TD_PAUSE_ACQ= Table Option" on page 1087

TD_TRACE_LEVEL=, TD_TRACE_LEVEL_INF= Table Options

Specifies the trace levels for driver tracing. TD_TRACE_LEVEL sets the primary trace level. TD_TRACE_LEVEL_INF sets the secondary trace level.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1028
Data source:	Teradata

Syntax

TD_TRACE_LEVEL= *trace-level*

Arguments

trace-level

specifies the type of diagnostic messages written by each instance of the driver to an external log file. The diagnostic trace function provides more detailed information (including the version number) in the log file to aid in problem tracking and diagnosis. The following trace levels are available:

TD_OFF	disables driver tracing. TD_OFF is the default setting for both driver tracing and infrastructure tracing. No external log file is produced unless this default is changed. Specifying TD_OFF for both driver tracing and infrastructure tracing is the same as disabling tracing.
TD_OPER	activates the tracing function for driver-specific activities. The absence of any value for the PauseAcq attribute means that the Update driver job executes both the acquisition phase and the application phase without pausing. This distributes all of the rows that were sent to the Teradata database during the acquisition phase to their final destination on the AMPs.
TD_OPER_CLI	activates the tracing function for CLIv2-related activities (interaction with the Teradata database).
TD_OPER_NOTIFY	activates the tracing function for activities related to the Notify feature.
TD_OPER_OPCOMMON	activates the tracing function for activities involving the opcommon library.
TD_OPER_ALL	activates the tracing function for all of the tracing activities.

Details

If the trace level is set to any value other than TD_OFF, an external log file is created for each instance of the driver.

The trace levels for infrastructure tracing should be used only when you are directed to by Teradata support. TD_OFF, which disables infrastructure tracing, should always be specified.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 1026

TD_TRACE_OUTPUT= Table Option

Specifies the name of the external file used for trace messages.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028

Data source: Teradata

Syntax

TD_TRACE_OUTPUT= *filename*

Arguments

filename
specifies the name of the external file to use for tracing. If a file with the specified name already exists, then the existing file is overwritten. The new filename is created with the name of the driver and a time stamp.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

TD_UNICODE_PASSTHRU= Table Option

Allows pass-through characters (PTCs) to be imported into and exported from Teradata.

Category: Bulk Loading

Alias: TPT_UNICODE_PASSTHRU=

Default: NO

Restrictions: This table option is not supported on the CAS server.
This attribute requires version 16.10 of the Teradata database, or later.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028

Data source: Teradata

Note: This option is available beginning with SAS 9.4M7.

Syntax

TD_UNICODE_PASSTHRU= YES | NO

Arguments

YES
import and export of PTCs is allowed.

NO
import and export of PTCs is not allowed. This is the default setting.

Details

The TD_UNICODE_PASSTHRU= session attribute controls use of PTCs for all connections that the Teradata TPT (Teradata Parallel Transporter) creates for LOAD, UPDATE, and STREAM operations. PTCs are the set of all possible Unicode characters minus the 6.0 BMP characters that are supported by Teradata and the set of non-characters. For more information, see "Loading Unicode Data with Unicode Pass Through" in Teradata database administration documentation.

The TD_UNICODE_PASSTHRU= table option overrides the setting of the TPT_UNICODE_PASSTHRU= connection option.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 1026

TD_WORK_TABLE= Table Option

Specifies a name for the TPT Work table.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified with the “BULKOPTS= Table Option” on page 1028
Data source:	Teradata

Syntax

TD_WORK_TABLE= *name-of-table*

Arguments

name-of-table
specifies a name for the TPT Work table. The default name for the table is *ttname*_WT, where *ttname* is the name of the target table. Table names

exceeding 27 characters are truncated to accommodate the three-character suffix. Therefore, you might want to specify a name for the table that will not be truncated. When truncation occurs, a message is written to the log.

Restriction This option is provided for the Update operator. It is ignored by the Load and Stream operators.

Details

The Work table is created in the default user (logon) database, optionally qualified with a schema, unless the TD_LOGDB= table option is used to specify a different location for utility tables.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026
- "TD_DROP_WORK_TABLE= Table Option" on page 1074
- "TD_LOGDB= Table Option" on page 1081

TD_WORKING_DB= Table Option

Specifies the database where the insert table is to be created.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified with the "BULKOPTS= Table Option" on page 1028
Data source:	Teradata

Syntax

TD_WORKING_DB= *database-name*

Arguments

database-name
specifies a database name. If this option is not specified, the table is created in the specified SCHEMA. If SCHEMA is not specified, the table is created in the default user (logon) database.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1026

THREADNUM= Table Option

Specifies the maximum number of I/O threads the SPD Engine can spawn for processing an SPD Engine data set.

Category:	Table Control
Restriction:	This table option is not supported on the CAS server.
Data source:	SPD Engine data set

Syntax

THREADNUM= *n*

Arguments

n

specifies the number of threads. The default is the value of the SPDEMAXTHREADS= system option, if set. Otherwise, the default is two times the number of CPUs on your computer.

Details

THREADNUM= enables you to specify the maximum number of I/O threads that the SPD Engine spawns for processing an SPD Engine data set. The THREADNUM= value applies to any of the following SPD Engine I/O processing:

- WHERE expression processing
- parallel index creation
- I/O requested by thread-enabled applications

Adjusting THREADNUM= enables the system administrator to adjust the level of CPU resources the SPD Engine can use for any process. For example, in a 64-bit processor system, setting THREADNUM=4 limits the process to, at most, four CPUs, thereby enabling greater throughput for other users or applications.

When THREADNUM= is greater than 1, parallel processing is likely to occur. Therefore, physical order might not be retained in the output. Setting THREADNUM=1 means that no parallel processing occurs

You can also use this option to explore scalability for WHERE expression evaluations.

SPDEMAXTHREADS=, a configurable system option, imposes an upper limit on the consumption of system resources and therefore constrains the THREADNUM= value.

TYPE= Table Option

Specifies the data set type for a specially structured SAS data set.

Category: Table Control

Restriction: This table option is not supported on the CAS server.

Data source: SAS data set, SPD Engine data set

Syntax

TYPE= *data-set-type*

Arguments

data-set-type
specifies the special type of the data set.

Details

Use the TYPE= table option to create a special data set in the proper format or to identify the special type of the data set.

Most data sets do not have a specified type. However, there are several specially structured data sets that are used by some SAS/STAT procedures. These SAS data sets contain special columns and rows, and they are usually created by SAS statistical procedures. Because most of the special data sets are used with SAS/STAT software, they are described in *SAS/STAT User's Guide*.

Other values are available in other SAS software products and are described in the appropriate documentation.

UNIQUESAVE= Table Option

Specifies to save observations with nonunique key values (the rejected observations) to a separate data set when inserting observations into data sets with unique indexes.

Valid in: INSERT statement

Category: User Control of SAS Index Usage

Restriction: This table option is not supported on the CAS server.

Interaction: Used in conjunction with SPDSUSDS= automatic macro variable

Data source: SPD Engine data set

Syntax

UNIQUESAVE= YES | NO

Arguments

YES
writes rejected observations to a separate, system-created table that can be accessed by a reference to the macro variable SPDSUSDS=.

NO
does not write rejected observations to a separate table (that is, ignores nonunique key values).

Details

When observations are inserted into a data set that has a unique index, the rejected observations are ignored. With UNIQUESAVE=YES, the rejected observations are saved to a separate data set whose name is stored in the macro variable SPDSUSDS. You can specify the macro variable in place of the data set name to identify the rejected observations.

USER= Table Option

Specifies the HDFS user name.

Category: Bulk Loading

Alias: USERID=, UID=

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1028.

Interaction: (Optional) Use with "PASSWORD= Table Option" on page 1061.

Data source: Impala

Syntax

USER=*HDFS-user*

Arguments

HDFS-user
specifies the name that represents the HDFS user. Quotation marks around the value are optional, unless the name contains spaces.

Example

```
BULKLOAD=YES BULKOPTS=(USER=HADOOP PWD="hdfs-password" CFG="config-
path")
```

See Also

Table Options:

- “BULKLOAD= Table Option” on page 1026
- “PASSWORD= Table Option” on page 1061

WHERENOINDEX= Table Option

Specifies a list of indexes to exclude when making WHERE expression evaluations.

Category:	User Control of SAS Index Usage
Restrictions:	This table option is not supported on the CAS server. Cannot be used with IDXWHERE=NO
Data source:	SPD Engine data set

Syntax

WHERENOINDEX= (*name1 name2*...)

Arguments

(*name1 name2*...)
specifies a list of index names that you want to exclude from use.

See Also

Table Options:

■ “IDXWHERE= Table Option” on page 1050

WRITE= Table Option Table Option

Assigns a WRITE password to a SAS file that prevents users from writing to the file or that enables access to a Write-protected file.

Category:	Table Control
Restriction:	This table option is not supported on the CAS server.
Data source:	SAS data set, SPD Engine data set
Note:	Check your log after this operation to ensure that the password values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in *SAS Programmer’s Guide: Essentials*.

Syntax

WRITE= *write-password*

Arguments

write-password
must be a valid SAS name.

Details

You can use this option to assign a password or to access a write-protected file.

Note: A SAS password does not control access to a SAS file beyond SAS. You should use the operating system-supplied utilities and file-system security controls in order to control access to SAS files outside SAS.

9

FedSQL and System Options

FedSQL Support for System Options

FedSQL does not provide any options that affect the processing of an entire SAS program or interactive SAS session from the time the option is specified until it is changed. Nor does FedSQL support SAS system options, with the exception of certain SAS invocation options. These SAS system options are described, where applicable, in this documentation.

PART 3

Appendixes

Appendix 1

Troubleshooting: Resolving FedSQL Errors

Resolving FedSQL Errors

Types of Errors

FedSQL writes errors to the SAS log when it is unable to complete a request. Most errors can be categorized as follows:

Usage errors
: Usage errors arise because of incorrect or invalid requests. FedSQL usage errors are typically caused by syntax, semantic, or logic errors. Syntax and semantic errors are detected at compile time and prevent compilation of a FedSQL statement. A FedSQL statement that successfully compiles might still contain logic errors that surface at execution time.

Resource errors
: Resource errors arise when resource service requests cannot be fulfilled. Various conditions lead to a resource service request failure. For example, the resource could be inaccessible, a resource threshold could be reached, or you could lack authorization to access the resource.

How to Resolve FedSQL Errors

To resolve errors:

1. Check the SAS log for error messages. These messages might help you understand the source of the problem or provide ideas to help you resolve the error.

   Here is an example that shows what the problem is. In this instance, a usage error occurred. The messages indicate that a problem occurred in the compilation phase and was either a syntax error or access violation. The problem was encountered at the word DRIP. In this instance, a review of the code showed that the word was encountered where a table option was expected. A quick check of the documentation shows that FedSQL does not support a DRIP table option. It does support a DROP= table option.

   ```
   ERROR: Compilation error.
   ERROR: Syntax error or access violation DRIP
   ```

2. In the SAS log, focus on the earliest error messages first. Sometimes a single problem leads to a cascade of failures and resolving the first error can also resolve subsequent errors.

3. If an error message indicates a resource error, check with your system administrator to monitor system resources. Examples of system resources include memory usage and disk space. If system resources are low, look for ways to increase resources or consider ways that your application could use them more efficiently.

   Here is an example that shows a resource error. The message indicates a memory threshold was reached.

   ```
   ERROR: Memory allocation failed.
   ```

4. If you have been unable to resolve the error using the previous three steps, contact SAS Technical Support. Here is a list of information you will need to supply: Four tips to remember when you contact SAS Technical Support.

   If the SAS log contains any error messages with internal error codes, provide the codes to your SAS Technical Support representative. Internal error codes are hexadecimal numbers that can help SAS more quickly determine the source of the problem. Here is an example of an error message with an internal error code.

   ```
   ERROR: FedSQL internal error code 0xECCC8C8BF7C049FD. Refer to FedSQL
    documentation for resolving errors.
   ```

Additional Information to Provide Technical Support for FedSQL Requests Submitted to a SAS Library

Beginning with SAS Viya 3.5, FedSQL supports a _DIAG option that can be used to gather additional information for SAS Technical Support. Your SAS Technical Support representative will request this information from you when you encounter an

internal error. The messages returned by the _DIAG option are intended for FedSQL developers and will not make sense to you.

Add the _DIAG option to the FedSQL request that failed. The following code shows how to add the option to the FedSQL procedure. The _DIAG option is specified as a procedure option.

```
proc fedsql _diag;
...FedSQL statements...
quit;
```

Other information that is helpful to provide to Technical Support includes the following:

- the number of tables involved in the failed request
- the number of columns and rows in each input table
- the data types used
- the data sources involved in the request
- the LIBNAME statement or connection string used to make the data source connection or connections

Appendix 2

Data Type Reference

Data Types for SAS Data Sets

The following table lists the data type support for a SAS data set.

The BINARY and VARBINARY data types are not supported for data definition.

For some data type definitions, the data type is mapped to CHAR, which is a Base SAS character data type, or DOUBLE, which is a Base SAS numeric data type. For data source-specific information about the SAS numeric and SAS character data types, see *SAS Language Reference: Concepts*.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about support for SAS data sets in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A7.1 *Mapping of FedSQL Data Types to Data Types Used by SAS Data Sets*

Data Type Definition Keyword [1]	SAS Data Set Data Type	Description	Data Type Returned
BIGINT[2]	DOUBLE	A 64-bit double precision, floating-point number. **Note:** There is potential for loss of precision.	DOUBLE
CHAR(*n*)	CHAR(*n*)	A fixed-length character string. **Note:** Cannot contain ANSI SQL null values.	CHAR(*n*)
DATE [3]	DOUBLE	A 64-bit double precision, floating-point number. by default, applies the DATE9 SAS format.	DOUBLE
DECIMAL| NUMERIC(*p*,*s*)[2]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
DOUBLE[2]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
FLOAT(*p*)[2]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE

Data Type Definition Keyword [1]	SAS Data Set Data Type	Description	Data Type Returned
INTEGER[2]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
NCHAR(*n*)	CHAR(*n*)	A fixed-length character string. By default, sets the encoding to Unicode UTF-8. [4]	CHAR(*n*)
NVARCHAR(*n*)	CHAR(*n*)	A fixed-length character string. By default, sets the encoding to Unicode UTF-8. [4]	CHAR(*n*)
REAL[2]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
SMALLINT[2]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
TIME(*p*)[3]	DOUBLE	A 64-bit double precision, floating-point number. By default, applies the TIME8 SAS format.	DOUBLE
TIMESTAMP(*p*)[3]	DOUBLE	A 64-bit double precision, floating-point number. By default, applies the DATETIME19.2 SAS format.	DOUBLE
TINYINT[2]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
VARCHAR(*n*)	CHAR(*n*)	A fixed-length character string. **Note:** Cannot contain ANSI SQL null values.	CHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 Do not apply date and time SAS formats to a numeric data type. For date and time values, use the DATE, TIME, or TIMESTAMP data types.

3 Because the values are stored as a double precision, floating-point number, you can use the values in arithmetic expressions.

4 UTF-8 is an MBCS encoding. Depending on the operating environment, UTF-8 characters are of varying width, from one to four bytes. The value for *n*, which is the maximum number of multibyte characters to store, is multiplied by the maximum length for the operating environment. Note that when you are transcoding, such as from UTF-8 to Wlatin2, the variable lengths (in bytes) might not be sufficient to hold the values, and the result is character data truncation.

Data Types for SPD Engine Data Sets

The following table lists the data type support for an SPD Engine data set.

The BINARY, DECIMAL, NUMERIC, NCHAR, NVARCHAR, and VARBINARY data types are not supported for data type definition.

For some data type definitions, the data type is mapped to CHAR, which is a Base SAS character data type, or DOUBLE, which is a Base SAS numeric data type. For

data source-specific information about the SAS numeric and SAS character data types, see *SAS Language Reference: Concepts*.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about support for SPD Engine data sets in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A7.2 *Mapping of FedSQL Data Types to Data Types Used by SPD Engine Data Sets*

Data Type Definition Keyword	SPD Data Set Data Type	Description	Data Type Returned
BIGINT[1]	DOUBLE	A 64-bit double precision, floating-point number. **Note:** There is potential for loss of precision.	DOUBLE
CHAR(*n*)	CHAR(*n*)	A fixed-length character string. **Note:** Cannot contain ANSI SQL null values.	CHAR(*n*)
DATE [2]	DOUBLE	A 64-bit double precision, floating-point number. By default, applies the DATE9 SAS format.	DOUBLE
DOUBLE[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
FLOAT(*p*)[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
INTEGER[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
REAL[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
SMALLINT[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
TIME(*p*)[2]	DOUBLE	A 64-bit double precision, floating-point number. By default, applies the TIME8 SAS format.	DOUBLE
TIMESTAMP(*p*)[2]	DOUBLE	A 64-bit double precision, floating-point number. By default, applies the DATETIME19.2 SAS format.	DOUBLE
TINYINT[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
VARCHAR(*n*)	CHAR(*n*)	A fixed-length character string. **Note:** Cannot contain ANSI SQL null values.	CHAR(*n*)

1 Do not apply date and time SAS formats to a numeric data type. For date and time values, use DATE, TIME, or TIMESTAMP data types.

2 Because the values are stored as double precision, floating-point numbers, you can use the values in arithmetic expressions.

Data Types for SPD Server Tables

The following table lists the data type support for an SPD Server table.

The BINARY, DECIMAL, NUMERIC, NCHAR, NVARCHAR, and VARBINARY data types are not supported for data type definition.

For some data type definitions, the data type is mapped to CHAR, which is a Base SAS character data type, or DOUBLE, which is a Base SAS numeric data type. For data source-specific information about the SAS numeric and SAS character data types, see *SAS Language Reference: Concepts*.

Note: This data source is not supported on the CAS server.

Table A7.3 *Mapping of FedSQL Data Types to Data Types Used by SPD Server Tables*

Data Type Definition Keyword	SPD Data Set Data Type	Description	Data Type Returned
BIGINT[1]	DOUBLE	A 64-bit double precision, floating-point number. **Note:** There is potential for loss of precision.	DOUBLE
CHAR(*n*)	CHAR(*n*)	A fixed-length character string. **Note:** Cannot contain ANSI SQL null values.	CHAR(*n*)
DATE [2]	DOUBLE	A 64-bit double precision, floating-point number. By default, applies the DATE9 SAS format.	DOUBLE
DOUBLE[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
FLOAT(*p*) [1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE

Data Type Definition Keyword	SPD Data Set Data Type	Description	Data Type Returned
INTEGER[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
REAL[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
SMALLINT[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
TIME(*p*)[2]	DOUBLE	A 64-bit double precision, floating-point number. By default, applies the TIME8 SAS format.	DOUBLE
TIMESTAMP(*p*)[2]	DOUBLE	A 64-bit double precision, floating-point number. By default, applies the DATETIME19.2 SAS format.	DOUBLE
TINYINT[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
VARCHAR(*n*)	CHAR(*n*)	A fixed-length character string. **Note:** Cannot contain ANSI SQL null values.	CHAR(*n*)

1 Do not apply date and time SAS formats to a numeric data type. For date and time values, use DATE, TIME, or TIMESTAMP data types.

2 Because the values are stored as double precision, floating-point numbers, you can use the values in arithmetic expressions.

Data Types for Amazon Redshift

The following table lists the data type support for an Amazon Redshift database.

The BINARY, TINYINT, and VARBINARY data types are not supported for data type definition.

For data source-specific information about Redshift data types, see the Amazon Redshift database documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about Amazon Redshift support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A7.4 *Mapping of FedSQL Data Types to Data Types Used by Amazon Redshift*

Data Type Definition Keyword [1]	Amazon Redshift Data Type	Description	Data Type Returned
BIGINT	BIGINT	A signed eight-byte integer.	BIGINT
[2]	BOOLEAN	A logical Boolean (true/false) value.	[2]
CHAR(*n*)	CHAR	A fixed-length character string.	CHAR(*n*)
DATE	DATE	Date value.	DATE
DECIMAL| NUMERIC(*p*,*s*)	DECIMAL, NUMERIC	A signed, fixed-point decimal number.	DECIMAL| NUMERIC(*p*,*s*)[4]
DOUBLE	DOUBLE PRECISION	A signed, double precision, floating-point number.	DOUBLE
FLOAT(*p*)	FLOAT	A signed, single precision or double precision, floating-point number.	FLOAT(*p*)
INTEGER	INTEGER	A regular signed, exact whole number.	INTEGER
NCHAR	NCHAR	A fixed-length Unicode character string.	CHAR
NVARCHAR	NVARCHAR	A varying-length Unicode character string.	VARCHAR
REAL	REAL	A signed, single precision floating-point number.	REAL
SMALLINT	SMALLINT	A small signed, exact whole number.	SMALLINT

Data Type Definition Keyword [1]	Amazon Redshift Data Type	Description	Data Type Returned
TIME(*p*)	[3]	A time value.	TIMESTAMP
TIMESTAMP(*p*)	TIMESTAMP	A date and time value.	TIMESTAMP(*p*)
[2]	TIMESTAMPZ	A date and time value with time zone.	[2]
VARCHAR(*n*)	VARCHAR	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The Amazon Redshift data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 Amazon Redshift does not support the TIME(*p*) data type. When you define a column of type TIME, a column of type TIMESTAMP is created instead.

4 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Aster

The following table lists the data type support for an Aster database.

The NCHAR, NVARCHAR, and TINYINT data types are not supported for data type definition.

For data source-specific information about Aster database data types, see the Aster database documentation.

Note: This data source is not supported on the CAS server.

Table A7.5 *Mapping of FedSQL Data Types to Data Types Used by Aster*

Data Type Definition Keyword	Aster Data Type	Description	Data Type Returned
BIGINT	BIGINT	A large signed, exact whole number.	BIGINT
BINARY(*n*)	BYTEA	A varying-length binary string.	BINARY(*n*)

Data Type Definition Keyword	Aster Data Type	Description	Data Type Returned
[1]	BOOL	One byte integral data type that can contain values 0, 1, or NULL.	[1]
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
DATE	DATE	A date value.	DATE
DECIMAL| NUMERIC(*p*,*s*)	NUMERIC(*p*,*s*)	A signed, fixed-point decimal number.	DECIMAL| NUMERIC(*p*,*s*)[2]
DOUBLE	FLOAT(*p*)	A signed, double precision, floating-point number.	DOUBLE
FLOAT(*p*)	FLOAT(*p*)	A signed, double precision, floating-point number.	FLOAT(*p*)
INTEGER	INTEGER	A regular signed, exact whole number.	INTEGER
REAL	REAL	A signed, single precision, floating-point number.	REAL
SMALLINT	SMALLINT	A small signed, exact whole number.	SMALLINT
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
VARBINARY(*n*)	BYTEA	A varying-length binary string.	VARBINARY(*n*)
[1]	TEXT	A varying-length large character string.	VARCHAR(*n*)
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string.	VARCHAR(*n*)

1 The Aster data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

2 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for DB2 under UNIX and PC Hosts

The following table lists the data type support for a DB2 database under UNIX and PC hosts.

The NCHAR, NVARCHAR, and TINYINT data types are not supported for data type definition.

For data source-specific information about the DB2 database data types, see the DB2 database documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about DB2 support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A7.6 *Mapping of FedSQL Data Types to Data Types Used by DB2 under UNIX and PC Hosts*

Data Type Definition Keyword[1]	DB2 Data Type	Description	Data Type Returned
BIGINT	BIGINT	A large signed, exact whole number.	BIGINT
BINARY(*n*)	CHAR(*n*) FOR BIT DATA	A fixed-length binary string.	BINARY(*n*)
[2]	BLOB(*n*[K|M|G])	A varying-length binary large object string.	[2]
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
[2]	CLOB(*n*[K|M|G])	A varying-length character large object string.	[2]
DATE	DATE	A date value.	DATE
[2]	DBCLOB(*n*[K|M|G])	A varying-length double-byte character large object.	[2]
DECIMAL| NUMERIC(*p*,*s*)	DECIMAL| NUMERIC(*p*,*s*)	A signed, fixed-point decimal number.	DECIMAL| NUMERIC(*p*,*s*)[3]

Data Type Definition Keyword[1]	DB2 Data Type	Description	Data Type Returned
DOUBLE	DOUBLE	A signed, double precision, floating-point number.	DOUBLE
FLOAT(*p*)	FLOAT(*p*)	A signed, single precision or double precision, floating-point number.	FLOAT(*p*)
[2]	GRAPHIC(*n*)	A fixed-length graphic string.	[2]
INTEGER	INTEGER	A regular signed, exact whole number.	INTEGER
[2]	LONG VARCHAR [FOR BIT DATA]	A varying-length character or binary string.	[2]
[2]	LONG VARGRAPHIC(*n*)	A varying-length graphic string.	[2]
REAL	REAL	A signed, single precision, floating-point number.	REAL
SMALLINT	SMALLINT	A small signed, exact whole number.	SMALLINT
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
VARBINARY(*n*)	VARCHAR(*n*) FOR BIT DATA	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string.	VARCHAR(*n*)
[2]	VARGRAPHIC(*n*)	A varying-length graphic string	[2]

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The DB2 data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Google BigQuery

The following table lists the data type support for a Google BigQuery project. SAS/ACCESS to Google BigQuery uses the GoLang BiqQuery API to communicate with Google BigQuery.

The FedSQL DECIMAL/NUMERIC(p,s) data type is not supported for data type definition.

The Google BigQuery NUMERIC, ARRAY, STRUCT, and GEOGRAPHY data types are not supported.

For data source-specific information about Google BigQuery data types, see the Google BigQuery documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about Google BigQuery support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A7.7 *Mapping of FedSQL Data Types to Data Types Used by Google BigQuery*

Data Type Definition Keyword	BigQuery Data Type	BigQuery Description	Data Type Returned
BIGINT	INT64	A large signed, exact whole number, with a range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.	BIGINT
BINARY	BYTES	A varying-length binary data.	VARBINARY[2]
[1]	BOOL	A Boolean value (True or False) that is case insensitive..	[1]
CHAR(*n*)	STRING	A varying-length Unicode character string.	VARCHAR[3]
DATE	DATE	A date value from 0001-01-01 to 9999-12-31.	DATE
DOUBLE	FLOAT64	Double precision, floating-point number.	DOUBLE
FLOAT	FLOAT64	Double precision, floating-point number.	DOUBLE
INTEGER	INT64	A regular signed, exact whole number.	BIGINT
NCHAR	STRING	A fixed-length Unicode character string.	VARCHAR

Data Type Definition Keyword	BigQuery Data Type	BigQuery Description	Data Type Returned
NVARCHAR	STRING	A varying-length Unicode character string.	VARCHAR
SMALLINT	INT64	A small signed, exact whole number	BIGINT
TIME(*p*)	TIME	A time value in hours, minutes, and seconds, in the range 00:00:00 to 23:59:59.999999.	TIME(*p*)
[1]	TIMESTAMP	A date and time value.	[1]
TIMESTAMP(*p*)[4]	DATETIME	A date and time value with microsecond precision, independent of time zone. Supported values are 0001-01-01 00:00:00 to 9999-12-31 23:59:59.999999.	TIMESTAMP(*p*)
TINYINT	INT64	A very small signed, exact whole number.	BIGINT
VARBINARY(*n*)	BYTES	A varying-length binary string.	VARBINARY [2]
VARCHAR(*n*)	STRING	A varying-length Unicode character string.	VARCHAR[3]

1 The BigQuery data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

2 Length is determined by the MAX_BINARY_LEN= LIBNAME option. If MAX_BINARY_LEN= is not set, the default length is 2,000 bytes.

3 The default length is 16,384 characters unless the MAX_CHAR_LEN= LIBNAME option or SCANSTRINGCOLUMNS table option is set.

4 Beginning with the April 2021 release of SAS/ACCESS software for SAS 9.4M7 and SAS Viya 3.5, the TIMESTAMP(*p*) data type maps to the Google BigQuery DATETIME data type. In earlier software releases, TIMESTAMP(*p*) maps to the BigQuery TIMESTAMP(*p*) data type.

Data Types for Greenplum

The following table lists the data type support for a Greenplum database.

The BINARY, NCHAR, NVARCHAR, and TINYINT data types are not supported for data type definition.

For data source-specific information about Greenplum data types, see the Greenplum database documentation.

Note: This data source is not supported on the CAS server.

Table A7.8 Mapping of FedSQL Data Types to Data Types Used by Greenplum

Data Type Definition Keyword[1]	Greenplum Data Type	Description	Data Type Returned
BIGINT	INT8	A large signed, exact whole number.	BIGINT
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
DATE	DATE	A date value.	DATE
DECIMAL| NUMERIC(*p*,*s*)	DECIMAL(*p*,*s*)	A signed, fixed-point decimal number.	DECIMAL| NUMERIC(*p*,*s*)[3]
DOUBLE	DOUBLE	A floating-point number.	DOUBLE
FLOAT(*p*)	FLOAT8(*p*)	A floating-point number.	FLOAT(*p*)
INTEGER	INTEGER	A regular signed, exact whole number.	INTEGER
REAL	REAL	A floating-point number.	REAL
SMALLINT	INT8	A small signed, exact whole number.	SMALLINT
TIME(*p*)	TIME(*p*)	A time value in hours, minutes, and seconds.	TIME(*p*)[2]
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
VARBINARY(*n*)	BYTEA	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 Due to the ODBC-style interface that is used to communicate with the Greenplum server, fractional seconds are lost in the data transfer from server to client.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for HAWQ

The following table lists the data type support for a HAWQ database.

The BINARY, NCHAR, NVARCHAR, and TINYINT data types are not supported for data type definition.

For data source-specific information about HAWQ data types, see the HAWQ database documentation.

Note: This data source is not supported on the CAS server.

Table A7.9 Mapping of FedSQL Data Types to Data Types Used by HAWQ

Data Type Definition Keyword [1]	HAWQ Data Type	Description	Data Type Returned
BIGINT	INT8	A large signed, exact whole number.	BIGINT
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
DATE	DATE	A date value.	DATE
DECIMAL| NUMERIC(*p*,*s*)	DECIMAL(*p*,*s*)	A signed, fixed-point decimal number.	DECIMAL| NUMERIC(*p*,*s*)[3]
DOUBLE	DOUBLE	A floating-point number.	DOUBLE
FLOAT(*p*)	FLOAT8(*p*)	A floating-point number.	FLOAT(*p*)
INTEGER	INTEGER	A regular signed, exact whole number.	INTEGER
REAL	REAL	A floating-point number.	REAL
SMALLINT	INT8	A small signed, exact whole number.	SMALLINT
TIME(*p*)	TIME(*p*)	A time value in hours, minutes, and seconds.	TIME(*p*)[2]
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
VARBINARY(*n*)	BYTEA	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 Due to the ODBC-style interface that is used to communicate with the HAWQ server, fractional seconds are lost in the data transfer from server to client.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for HDMD

The following table lists the data type support for HDMD.

The BINARY, NCHAR, and NVARCHAR data types are not supported for data type definition.

Note: This data source is not supported on the CAS server.

Table A7.10 Mapping of FedSQL Data Types to Data Types Used by HDMD Tables

Data Type Definition Keyword	HDMD Data Type	Description	Data Type Returned
BIGINT	BIGINT	A large signed, exact whole number, with a precision of 19 digits. The range of integers is -9,223,372,036,854,775 ,807 to 9,223,372,036,854,775, 807.	BIGINT
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
DATE	DATE	Date value.	DATE
DECIMAL/ NUMERIC(*p*,*s*)	DECIMAL	A fixed-point decimal number, with 38 digits precision.	DECIMAL(*p*,*s*)[1]
DOUBLE	DOUBLE	A signed, approximate, double-precision, floating-point number. **Note:** Supports SAS missing values	DOUBLE
FLOAT	DOUBLE	A signed, approximate, double-precision, floating-point number. **Note:** Supports SAS missing values	DOUBLE
INTEGER	INTEGER	A regular size signed, exact whole number, with a precision of 10	INTEGER

Data Type Definition Keyword	HDMD Data Type	Description	Data Type Returned
		digits. The range of integers is -2,147,483,647 to 2,147,483,647.	
REAL	REAL	A signed, approximate, single-precision, floating-point number. **Note:** Supports SAS missing values	REAL
SMALLINT	SMALLINT	A small signed, exact whole number, with a precision of five digits. The range of integers is -32,767 to 32,767.	SMALLINT
TIME(*p*)	TIME[(*p*)]	A time value with optional precision.	TIME[(*p*)]
TIMESTAMP(*p*)	TIMESTAMP[(*p*)]	A date and time value with optional precision.	TIMESTAMP[(*p*)]
TINYINT	TINYINT	A very small signed, exact whole number, with a precision of three digits. The range of integers is -127 to 127.	TINYINT
VARCHAR(*n*)	VARCHAR(n)	A varying-length character string data.	VARCHAR(*n*)

1 The DECIMAL data type is handled as double precision.

Data Types for Hive

The following table lists the data type support for Hive. Hive database requirements can vary, depending on your SAS software release. See the Support for Hadoop for information about SAS 9.4 database requirements.

The NCHAR and NVARCHAR data types are not available for data definition. The VARBINARY data type is available for data definition beginning with SAS 9.4M3.

The Hive complex data types can be read, beginning with SAS 9.4M3.

For data source-specific information about Hive data types, see the Hive database documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about Hive support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A7.11 *Mapping of FedSQL Data Types to Data Types Used by Hive*

Data Type Definition Keyword	Hive Data Type	Description	Data Type Returned
[1, 6]	ARRAY<*data-type*>	An array of integers (indexable lists).	STRING[10]
BIGINT	BIGINT	A signed eight-byte integer, from -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807.	BIGINT
BINARY(*n*)	BINARY[2]	A varying-length binary string up to 32K.	BINARY
[1]	BOOLEAN	A textual true or false value.	TINYINT
CHAR(*n*)	CHAR(n)[2]	A character string up to 255 characters. If you specify CHAR with a value greater than 255 characters, the column is created as VARCHAR(n) instead.	CHAR
DATE	DATE[3, 4]	An ANSI SQL date type.	DATE
DECIMAL/ NUMERIC(*p*,*s*)	DECIMAL[2]	A fixed-point decimal number, with 38 digits precision.	DECIMAL(*p*,*s*)[5]
DOUBLE	DOUBLE	An eight-byte, double-precision floating-point number.	DOUBLE
FLOAT(*p*)	FLOAT	A four-byte, single-precision floating-point number.	DOUBLE
INTEGER	INTEGER	A regular size signed, exact whole number, with a precision of 10 digits. The range of integers is -2,147,483,647 to 2,147,483,647.	INTEGER
[1, 6]	MAP<*primitive-type*, *data-type*>	An associative array of key-value pairs.	STRING[10]
REAL	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
SMALLINT	SMALLINT	A signed two-byte integer, from -32,767 to 32,767.	SMALLINT

Data Type Definition Keyword	Hive Data Type	Description	Data Type Returned
[1, 6]	STRING	A varying-length character string.	VARCHAR(n)[8]
[1, 6]	STRUCT<*col-name*: *data-type*>	A structure with established column elements and data types. Column elements and data types are mapped using a double-dot (:) notation.	STRING[10]
TIME(*p*)	[9]	A time value.	STRING
TIMESTAMP(*p*)	TIMESTAMP	A UNIX timestamp with optional nanosecond precision.	TIMESTAMP[(*p*)]
TINYINT	TINYINT	A signed one-byte integer, from -127 to 127.	TINYINT
[1, 6]	UNION<*data-type*, *data-type-n*>	A type that can hold one of several specified data types.	STRING[10]
VARCHAR(*n*)[7]	VARCHAR(*n*)	A varying-length character string.	STRING
VARBINARY	BINARY[2]	A varying-length binary string up to 32K.	BINARY

1 The Hive data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

2 Full support for this data type is available in Hive 0.13 and later. In Hadoop environments that use earlier Hive versions (which do not support the CHAR and DECIMAL types), columns defined as CHAR are mapped to VARCHAR. Columns that are defined as DECIMAL are not supported. In Hadoop environments that use Hive versions earlier than Hive 0.13, BINARY columns can be created but not retrieved.

3 Full support for this data type is available in Hive 0.12 and later. In Hadoop environments that use earlier Hive versions (which do not support the DATE type), when the DATE data type is used for data definition, the DATE type is mapped to a STRING column with SASFMT TableProperties. Any SASFMT TableProperties that are defined on STRING columns are applied when reading Hive, allowing the STRING columns to be treated as DATE columns. For more information about SASFMT TableProperties, see “SAS Table Properties for Hive and HADOOP” in *SAS/ACCESS for Relational Databases: Reference*.

4 The supported date values are between October 15, 1582 and December 31, 9999. Date values containing years earlier than 1582 return an error. Date values later than 9999 are read back as null values.

5 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

6 String and Hive complex types are loaded as VARCHAR(*n*), whose length is determined by the DBMAX_TEXT= data source connection option.

7 VARCHAR columns with values greater than 65,535 characters are created as STRING.

8 SASFMT TableProperties are applied when reading STRING columns.

9 Hive does not support the TIME(p) data type. When data is read from Hive, STRING columns that have SASFMT TableProperties defined that specify the SAS TIME8. format are converted to the TIME(*p*) data type. When the TIME type is used for data definition, it is mapped to a STRING column with SASFMT TableProperties. Fractional seconds are not preserved. For more information about SASFMT TableProperties, see “SAS Table Properties for Hive and HADOOP” in *SAS/ACCESS for Relational Databases: Reference*.

10 The complex types ARRAY, MAP, STRUCT, and UNION are read as their STRING representation of the underlying complex type. ARRAY values are read back within brackets, for example: [1, 2, 4]. STRUCT and MAP values are read back within braces, for example: {"firstname":"robert","nickname":"bob"}.

Data Types for Impala

The following table lists the data type support for Impala. Impala version 2.0 and later are supported, running on CDH 5.1 and later.

The BINARY, DECIMAL(*p*,*s*)/NUMERIC, NCHAR, NVARCHAR, and VARBINARY data types are not available for data definition.

For data source-specific information about Impala data types, see the vendor documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about Impala support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A7.12 *Mapping of FedSQL Data Types to Data Types Used by Impala*

Data Type Definition Keyword	Impala Data Type	Description	Data Type Returned
BIGINT	BIGINT	A signed eight-byte integer, from -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807.	BIGINT
CHAR(*n*)	CHAR[1]	A fixed-length character string up to 255 characters.	CHAR
DATE	[2]	An ANSI SQL date type.	TIMESTAMP
DOUBLE	DOUBLE	An eight-byte, double-precision floating-point number.	DOUBLE
FLOAT(*p*)	FLOAT	A four-byte, single-precision floating-point number.	DOUBLE
INTEGER	INT	A signed four-byte integer, from -2,147,483,647 to 2,147,483,647.	INTEGER
REAL	DOUBLE	An eight-byte, double-precision floating-point number.	DOUBLE

Data Type Definition Keyword	Impala Data Type	Description	Data Type Returned
SMALLINT	SMALLINT	A signed two-byte integer, from -32,767 to 32,767.	SMALLINT
TIME	[2]	A time value.	TIMESTAMP
TIMESTAMP	TIMESTAMP	A UNIX timestamp with optional nanosecond precision.	TIMESTAMP
TINYINT	TINYINT	A signed one-byte integer, from -127 to 127.	TINYINT
VARCHAR(*n*)	VARCHAR[1]	A varying-length character string.	VARCHAR

1 Support for this data type is available in CDH 5.2 and later. In environments that use earlier CDH versions, which do not support the CHAR and VARCHAR types, columns defined as CHAR or VARCHAR are mapped to STRING.

2 Impala does not support this data type. When a DATE or TIME column is defined, it is created as a column of type TIMESTAMP.

Data Types for JDBC

The following table lists the data type support for JDBC. JDBC 4.1 and later is supported.

For data source-specific information about JDBC data types, see the database documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about support for JDBC databases in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A7.13 *Mapping of FedSQL Data Types to Data Types Supported by JDBC*

Data Type Definition Keyword	JDBC SQL Identifier	Description	Data Type Returned
BIGINT	SQL_BIGINT	A large signed, exact whole number.	BIGINT
BINARY(*n*)	SQL_BINARY	A fixed-length binary string.	BINARY(*n*)

Data Type Definition Keyword	JDBC SQL Identifier	Description	Data Type Returned
[1]	SQL_BIT	Single bit binary data.	[1]
CHAR(*n*)[2]	SQL_CHAR	A fixed-length character string.	CHAR(*n*)
DATE	SQL_TYPE_DATE	A date value.	DATE
DECIMAL| NUMERIC(*p*,*s*)	SQL_DECIMAL| SQL_NUMERIC	A signed, fixed-point decimal number.	DECIMAL| NUMERIC(*p*,*s*)[3]
DOUBLE	SQL_DOUBLE	A signed, double precision, floating-point number.	DOUBLE
FLOAT(*p*)	SQL_FLOAT	A signed, approximate, floating-point number.	FLOAT(*p*)
[1]	SQL_GUID	A globally unique identifier.	[1]
INTEGER	SQL_INTEGER	A regular signed, exact whole number.	INTEGER
[1]	SQL_INTERVAL	Intervals between two years, months, days, dates or times.	[1]
[1]	SQL_LONGVARBINAR Y	A varying-length binary string.	[1]
[1]	SQL_LONGVARCHAR	A varying-length Unicode character string.	[1]
NCHAR(*n*)	SQL_WCHAR	A fixed-length Unicode character string.	NCHAR(*n*)
NVARCHAR(*n*)	SQL_WVARCHAR	A varying-length Unicode character string.	NVARCHAR(*n*)
REAL	SQL_REAL	A signed, single precision, floating-point number.	REAL
SMALLINT	SQL_SMALLINT	A small signed, exact whole number.	SMALLINT
TIME(*p*)	SQL_TYPE_TIME	A time value.	TIME(*p*)

Data Type Definition Keyword	JDBC SQL Identifier	Description	Data Type Returned
TIMESTAMP(*p*)	SQL_TYPE_TIMESTAMP	A date and time value.	TIMESTAMP(*p*)
TINYINT	SQL_TINYINT	A very small signed, exact whole number.	TINYINT
VARBINARY(*n*)	SQL_VARBINARY	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)[2]	SQL_VARCHAR	A varying-length character string.	VARCHAR(*n*)
[1]	SQL_WLONGVARCHAR	A varying-length Unicode character string.	[1]

1 The JDBC SQL data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.
2 When you use the CHAR(n) or VARCHAR(n) data type to store multibyte data, you must specify the encoding in the CLIENT_ENCODING= connection option. Or, to avoid having to set the encoding, use the NCHAR or NVARCHAR data types for multibyte data instead.
3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for MDS

The following table lists the data type support for the in-memory database.

Note: This data source is not supported on the CAS server.

Table A7.14 *Mapping of FedSQL Data Types to Data Types Used by MDS Tables*

Data Type Definition Keyword [2]	MDS Data Type	Description	Data Type Returned
BIGINT	BIGINT	A 64-bit, signed integer.	BIGINT
BINARY(*n*)	BINARY(*n*)	A fixed-length binary data.	BINARY(*n*)
CHAR(*n*)	CHAR(*n*)	A fixed-length character string data.	CHAR(*n*)
DATE	DATE	Date value.	DATE

Data Type Definition Keyword [2]	MDS Data Type	Description	Data Type Returned
DECIMAL| NUMERIC(*p*,*s*)	NUMERIC(*p*,*s*)	Precision scale numeric.	DECIMAL/ NUMERIC(*p*,*s*)[3]
DOUBLE	DOUBLE	An eight-byte IEEE floating-point value. **Note:** Supports ANSI SQL null values	DOUBLE
FLOAT(*p*)	DOUBLE	An eight-byte IEEE floating-point value. **Note:** Supports ANSI SQL null values	DOUBLE
INTEGER	INTEGER	32-bit, signed integer.	INTEGER
NCHAR(*n*)	NCHAR(*n*)	A fixed-length Unicode character string.	NCHAR(*n*)
NVARCHAR(*n*)	NVARCHAR(*n*)	A varying-length Unicode character string.	NVARCHAR(*n*)
REAL	DOUBLE	An eight-byte IEEE floating-point value. **Note:** Supports ANSI SQL null values	DOUBLE
SMALLINT	INTEGER	32-bit, signed integer.	INTEGER
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
TINYINT	INTEGER	32-bit, signed integer.	INTEGER
[1]	UBIGINT	A 64-bit, unsigned integer.	BIGINT
[1]	UINTEGER	32-bit, unsigned integer.	INTEGER
VARCHAR(*n*)	VARCHAR(n)	A varying-length character string data.	VARCHAR(*n*)

Data Type Definition Keyword [2]	MDS Data Type	Description	Data Type Returned
VARBINARY(n)	VARBINARY(n)	A varying-length binary data.	VARBINARY(n)

1 The MDS SQL data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

2 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Microsoft SQL Server

The following table lists the data type support for a Microsoft SQL Server database. SAS supports SQL Server 2008 and later.

For data source-specific information about the Microsoft SQL Server data types, see the Microsoft SQL Server database documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about Microsoft SQL Server support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A7.15 *Mapping of FedSQL Data Types to Data Types Used by Microsoft SQL Server*

Data Type Definition Keyword[1]	SQL Server SQL Identifier	Description	Data Type Returned
BIGINT	BIGINT	A large signed, exact whole number.	BIGINT
BINARY(*n*)	BINARY(*n*)	A fixed-length binary string.	BINARY(*n*)
[2]	BIT	Single bit binary data.	[2]
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
DATE	DATE	A date value.	DATE
[2]	DATETIME	Date combined with time of day with fractional	TIMESTAMP(*p*)

Data Type Definition Keyword[1]	SQL Server SQL Identifier	Description	Data Type Returned
		seconds that is based on a 24-hour clock.	
[2]	DATETIME2(*p*)	An extension of DATETIME with larger date range, a larger default fractional precision, and optional user-specified precision.	TIMESTAMP(*p*)
[2]	DATETIMEOFFSET(*p*)	Date/time value with time zone awareness.	[2]
DECIMAL| NUMERIC(*p*,*s*)	DECIMAL| NUMERIC(*p*,*s*)	A signed, fixed precision and scale numbers.	DECIMAL| NUMERIC(*p*,*s*)[3]
DOUBLE	DOUBLE	A signed, double precision, floating-point number.	DOUBLE
FLOAT(*p*)	FLOAT(*n*)	A signed, approximate, floating-point number.	FLOAT(*p*)
[2]	IMAGE	A varying-length binary string.	[2]
INTEGER	INTEGER	A regular signed, exact whole number.	INTEGER
[2]	MONEY	An eight-byte currency value.	DECIMAL(19,4)[3]
NCHAR(*n*)	NCHAR	A fixed-length Unicode character string.	NCHAR(*n*)
NVARCHAR(*n*)	NVARCHAR	A varying-length Unicode character string.	NVARCHAR(*n*)
REAL	REAL	A signed, single precision, floating-point number.	REAL
[2]	SMALLDATETIME	A date and time value without fractional seconds.	TIMESTAMP(*p*)
SMALLINT	SMALLINT	A small signed, exact whole number.	SMALLINT

Data Type Definition Keyword[1]	SQL Server SQL Identifier	Description	Data Type Returned
[2]	SMALLMONEY	A four-byte currency value.	DECIMAL(10,4)[3]
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)
TIMESTAMP(*p*)	BINARY(8)	A date and time value.	TIMESTAMP(*p*)
TINYINT	TINYINT	A very small signed, exact whole number.	TINYINT
[2]	UNIQUEIDENTIFIER	A globally unique identifier.	CHAR(36)
VARBINARY(*n*)	VARBINARY	A varying-length binary string.	VARBINARY(*n*)
[2]	VARBINARY(MAX)	A varying-length binary string. Maximum 2GB	[2]
VARCHAR(*n*)	VARCHAR(*n*), TEXT	A varying-length character string.[2]	VARCHAR(*n*)
[2]	VARCHAR(MAX)	A varying-length Unicode character string. Maximum 2GB	[2]

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The Microsoft SQL Server data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for MongoDB

The following table lists the data type support for a MongoDB non-relational database. MongoDB 3.6 and later is supported.

The MongoDB types Timestamp, Min key, and Max key are internal to MongoDB. Therefore, they are not read by SAS.

The FedSQL BINARY, DATE, DECIMAL, FLOAT, NCHAR, NVARCHAR, REAL, SMALLINT, TINYINT, and VARBINARY data types are not supported for data definition.

For data source-specific information about MongoDB data types, see the MongoDB documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about MongoDB support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A7.16 *Mapping of FedSQL Data Types to Data Types Used by MongoDB*

Data Type Definition Keyword	MongoDB Data Type	Description	Data Type Returned
[2]	Array	Array.	[4]
BIGINT	int	A regular 64-bit exact whole number.	BIGINT
[1]	BinData	Binary data.	VARCHAR[5]
[1]	Bool	Boolean value.	INTEGER
CHAR(*n*)	String	A fixed-length UTF-8 character string.	CHAR(*n*)
[1]	Date	A date and time value.	TIMESTAMP
[5]	Decimal	Decimal128.	DOUBLE
DOUBLE	Double	A signed double-precision, floating-point number.	DOUBLE
INTEGER	Int	A regular 32-bit exact whole number.	INTEGER
[1]	Javascript	Javascript string.	[4]
[1]	Long	A large signed, exact whole number.	BIGINT
[1]	Null	Null value.	VARCHAR
[1]	Object	JSON object.	[4]
[1]	ObjectId	Object identifier.	VARCHAR
[1]	Regex	A regular expression string.	VARCHAR
TIMESTAMP	Date	A date and time value.	TIMESTAMP

Data Type Definition Keyword	MongoDB Data Type	Description	Data Type Returned
VARCHAR	String	A varying-length UTF-8 character string.	VARCHAR, CHAR(*n*)[3]

1 The MongoDB data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.
2 Arrays can be created as child tables. For more information, see information about MongoDB in *SAS/ACCESS for Nonrelational Databases: Reference*.
3 You can customize the default MongoDB schema to use CHAR(n). For more information, see the documentation for MongoDB in *SAS/ACCESS for Nonrelational Databases: Reference*.
4 The array and object types are modeled as separate child tables. For more information, see the documentation for MongoDB in *SAS/ACCESS for Nonrelational Databases: Reference*. The Javascript type is mapped to VARCHAR.
5 Contains a hexadecimal string representation of binary data.

Data Types for MySQL

The following table lists the data type support for a MySQL database. Support for MySQL begins with MySQL 5.1 and, beginning in SAS Viya 3.5, includes MySQL 8 server.

The NCHAR, NVARCHAR, REAL, and VARBINARY data types are not supported for data type definition.

For data source-specific information about MySQL data types, see the MySQL database documentation.

Note: This data source is not supported on the CAS server.

Table A7.17 *Mapping of FedSQL Data Types to Data Types Used by MySQL*

Data Type Definition Keyword[1]	MySQL Data Type	Description	Data Type Returned
BIGINT	BIGINT	A large signed, exact whole number.	BIGINT
BINARY(*n*)	BINARY(*n*)	A fixed-length binary string.	BINARY(*n*)
[2]	BLOB	A varying-length binary large object string.	[2]
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
DATE	DATE	A date value.	DATE
[2]	DATETIME	A date and time value.	[2]

Data Type Definition Keyword[1]	MySQL Data Type	Description	Data Type Returned
DECIMAL| NUMERIC(*p*,*s*)	DECIMAL(*p*,*s*)	A signed, fixed-point decimal number.	DECIMAL(*p*,*s*)[3]
DOUBLE	DOUBLE	A signed, double precision, floating-point number.	DOUBLE
[2]	ENUM(*values*)	A character value from a list of allowed values.	[2]
FLOAT(*p*)	FLOAT(*p*)	A signed, single precision or double precision, floating-point number.	FLOAT(*p*)
INTEGER	INT	A regular signed, exact whole number.	INTEGER
[2]	JSON	A varying-length character string in JSON format.	VARCHAR
[2]	LONGBLOB	A varying-length binary data.	[2]
[2]	LONGTEXT	A varying-length character string.	[2]
[2]	MEDIUMBLOB	A varying-length binary data.	[2]
[2]	MEDIUMINT	A regular signed, exact whole number.	[2]
[2]	MEDIUMTEXT	A varying-length character string.	[2]
[2]	SET(*values*)	Character values from a list of allowed values.	[2]
SMALLINT	SMALLINT	A small signed, exact whole number.	SMALLINT
[2]	TEXT	A varying-length text data.	[2]
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
[2]	TINYBLOB	A varying-length binary large object string.	[2]
TINYINT	TINYINT	A very small signed, exact whole number.	TINYINT
[2]	TINYTEXT	A varying-length text data.	[2]

Data Type Definition Keyword[1]	MySQL Data Type	Description	Data Type Returned
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The MySQL data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Netezza

The following table lists the data type support for a Netezza database.

The BINARY and VARBINARY data types are not supported for data type definition.

For data source-specific information about Netezza data types, see the Netezza database documentation.

Note: This data source is not supported on the CAS server.

Table A7.18 *Mapping of FedSQL Data Types to Data Types Used by Netezza*

Data Type Definition Keyword[1]	Netezza Data Type	Description	Data Type Returned
BIGINT	BIGINT	A large signed, exact whole number.	BIGINT
CHAR(*n*)	CHAR(*n*)	A fixed-length character string data.	CHAR(*n*)
DATE	DATE	A date value.	DATE
DECIMAL| NUMERIC(*p*,*s*)	DECIMAL(*p*,*s*)	A fixed-point decimal number.	DECIMAL(*p*,*s*)[3]
DOUBLE	DOUBLE	A floating-point number.	DOUBLE
FLOAT(*p*)	FLOAT(*p*)	A 64-bit double precision, floating-point number.	FLOAT(*p*)
INTEGER	INTEGER	A large integer.	INTEGER

Data Type Definition Keyword[1]	Netezza Data Type	Description	Data Type Returned
NCHAR(*n*)	NCHAR(*n*)	A fixed-length Unicode character string.	NCHAR(*n*)
NVARCHAR(*n*)	NVARCHAR(*n*)	A varying-length Unicode character string.	NVARCHAR(*n*)
REAL	REAL	A floating-point number.	REAL
SMALLINT	SMALLINT	A small integer.	SMALLINT
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)[2]
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
TINYINT	BYTEINT	A tiny integer.	TINYINT
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string data.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 Due to the ODBC-style interface that is used to communicate with the Netezza server, fractional seconds are lost in the data transfer from server to client.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for ODBC

The following table lists the data type support for a data source that is compliant with ODBC.

For data source-specific information about ODBC SQL data types, see the specific ODBC data source documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about support for ODBC databases in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A7.19 Mapping of FedSQL Data Types to Data Types Supported by ODBC

Data Type Definition Keyword[1]	ODBC SQL Identifier	Description	Data Type Returned
BIGINT	SQL_BIGINT	A large signed, exact whole number.	BIGINT
BINARY(*n*)	SQL_BINARY	A fixed-length binary string.	BINARY(*n*)
[2]	SQL_BIT	Single bit binary data.	[2]
CHAR(*n*)[3]	SQL_CHAR	A fixed-length character string.	CHAR(*n*)
DATE	SQL_TYPE_DATE	A date value.	DATE
DECIMAL| NUMERIC(*p*,*s*)	SQL_DECIMAL| SQL_NUMERIC	A signed, fixed-point decimal number.	DECIMAL| NUMERIC(*p*,*s*)[4]
DOUBLE	SQL_DOUBLE	A signed, double precision, floating-point number.	DOUBLE
FLOAT(*p*)	SQL_FLOAT	A signed, approximate, floating-point number.	FLOAT(*p*)
[2]	SQL_GUID	A globally unique identifier.	[2]
INTEGER	SQL_INTEGER	A regular signed, exact whole number.	INTEGER
[2]	SQL_INTERVAL	Intervals between two years, months, days, dates or times.	[2]
[2]	SQL_LONGVARBINARY	A varying-length binary string.	[2]
[2]	SQL_LONGVARCHAR	A varying-length Unicode character string.	[2]
NCHAR(*n*)	SQL_WCHAR	A fixed-length Unicode character string.	NCHAR(*n*)
NVARCHAR(*n*)	SQL_WVARCHAR	A varying-length Unicode character string.	NVARCHAR(*n*)

Data Type Definition Keyword[1]	ODBC SQL Identifier	Description	Data Type Returned
REAL	SQL_REAL	A signed, single precision, floating-point number.	REAL
SMALLINT	SQL_SMALLINT	A small signed, exact whole number.	SMALLINT
TIME(*p*)	SQL_TYPE_TIME	A time value.	TIME(*p*)
TIMESTAMP(*p*)	SQL_TYPE_TIMESTAMP	A date and time value.	TIMESTAMP(*p*)
TINYINT	SQL_TINYINT	A very small signed, exact whole number.	TINYINT
VARBINARY(*n*)	SQL_VARBINARY	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)[3]	SQL_VARCHAR	A varying-length character string.	VARCHAR(*n*)
[2]	SQL_WLONGVARCHAR	A varying-length Unicode character string.	[2]

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The ODBC SQL data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 When you use the CHAR(*n*) or VARCHAR(*n*) data type to store multibyte data in a DB2, Greenplum, or Oracle database, you must specify the encoding in the CLIENT_ENCODING= connection option. Or, for Oracle only, to avoid having to set the encoding, use the NCHAR or NVARCHAR data types for multibyte data instead.

4 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Oracle

The following table lists the data type support for an Oracle database.

For data source-specific information about Oracle data types, see the Oracle database documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about

Oracle support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A7.20 *Mapping of FedSQL Data Types to Data Types Used by Oracle*

Data Type Definition Keyword[1]	Oracle Data Type	Description	Data Type Returned
BIGINT	NUMBER	A large signed, exact whole number.	BIGINT
BINARY(*n*)	RAW(*n*)	A fixed or varying-length binary string.	BINARY(*n*)
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
DATE	DATE	A date value.	TIMESTAMP(*p*)[3]
DECIMAL| NUMERIC(*p*,*s*)	NUMBER(*p*,*s*)	A signed, fixed-point decimal number.	DOUBLE[4]
DOUBLE	BINARY_DOUBLE	A signed, double precision floating-point number.	DOUBLE
FLOAT(*p*)	FLOAT(p)	A signed, double precision floating-point number.	DOUBLE
INTEGER	NUMBER(*p*,*s*)	A regular signed, exact whole number.	INTEGER
[2]	LONG	A varying-length character string data.	[2]
NCHAR(*n*)	NCHAR(*n*)	A fixed-length Unicode character string.	NCHAR(*n*)
NVARCHAR(*n*)	NVARCHAR(*n*)	A varying-length Unicode character string.	NVARCHAR(*n*)
REAL	FLOAT	A signed, single precision floating-point number.	REAL
SMALLINT	NUMBER	A small signed, exact whole number.	SMALLINT
TIME(*p*)	TIME(*p*)	A time value.	TIMESTAMP(*p*)[3]
TIMESTAMP(*p*)[5]	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
TINYINT	NUMBER	A very small signed, exact whole number.	TINYINT

Data Type Definition Keyword[1]	Oracle Data Type	Description	Data Type Returned
VARBINARY(*n*)	LONG RAW(*n*)	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)	VARCHAR2(*n*)[6]	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The Oracle data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The timestamp returned by the DATE and TIME data types can be changed to date and time values by using the DATEPART function with the PUT function.

4 The ORNUMERIC= connection argument and table option determine how numbers read from or inserted into the Oracle NUMBER column are treated. ORNUMERIC=YES, which is the default, indicates that non-integer values with explicit precision are treated as NUMERIC values.

5 The TIMESTAMP(*p*) data type is not available on z/OS.

6 The VARCHAR2(*n*) type is supported for up to 32,767 bytes if the Oracle version is 12c and the Oracle MAX_STRING_SIZE= parameter is set to EXTENDED.

Data Types for PostgreSQL

The following table lists the data type support for a PostgreSQL database.

The BINARY, NCHAR, NVARCHAR, TINYINT, and VARBINARY data types are not supported for data type definition.

For data source-specific information about PostgreSQL data types, see the PostgreSQL database documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about PostgreSQL support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A7.21 *Mapping of FedSQL Data Types to Data Types Used by PostgreSQL*

Data Type Definition Keyword[1]	PostgreSQL Data Type	Description	Data Type Returned
BIGINT	BIGINT	A large signed, exact whole number or a signed eight-byte integer.	BIGINT
[2]	BIGSERIAL	An auto-incrementing eight-byte integer.	[2]

Data Type Definition Keyword[1]	PostgreSQL Data Type	Description	Data Type Returned
[2]	BIT(*n*)	A fixed-length bit string.	[2]
[2]	BIT VARYING(*n*)	A varying-length bit string.	[2]
[2]	BOOLEAN	A logical Boolean (true/false) value.	[2]
[2]	BOX	A rectangular box on a plane.	[2]
[2]	BYTEA	A byte array.	[2]
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
[2]	CIDR	An IPv4 or IPv6 network address.	[2]
[2]	CIRCLE	A circle on a plane.	[2]
DATE	DATE	A date value.	DATE
DECIMAL| NUMERIC(*p*,*s*)	NUMERIC(*p*,*s*)	A signed, fixed-point decimal number.	DECIMAL| NUMERIC(*p*,*s*)[3]
DOUBLE	DOUBLE PRECISION	A signed, double precision, floating-point number.	DOUBLE
FLOAT(*p*)	FLOAT(*p*)	A signed, single precision or double precision, floating-point number.	FLOAT(*p*)
[2]	INET	An IPv4 or IPv6 host address.	[2]
INTEGER	INTEGER	A regular signed, exact whole number.	INTEGER
[2]	INTERVAL	A time span.	[2]
[2]	LINE	An infinite line on a plane.	[2]
[2]	LSEG	A line segment on a plane.	[2]
[2]	MACADDR	A Media Access Control address.	[2]
[2]	MONEY	A currency value.	[2]
[2]	PATH	A geometric path on a plane.	[2]
[2]	POINT	A geometric point on a plane.	[2]

Data Type Definition Keyword[1]	PostgreSQL Data Type	Description	Data Type Returned
[2]	POLYGON	A closed geometric path on a plane.	[2]
REAL	REAL	A signed, single precision floating-point number.	REAL
[2]	SERIAL	An auto-incrementing four-byte integer.	[2]
SMALLINT	SMALLINT	A small signed, exact whole number.	SMALLINT
[2]	SMALL SERIAL	An auto-incrementing two-byte integer.	[2]
[2]	TEXT	A varying-length character string.	[2]
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
[2]	TSQUERY	A text search query.	[2]
[2]	TSVECTOR	A text search document.	[2]
[2]	TXID_SNAPSHOT	A snapshot of a user-level transaction ID.	[2]
[2]	UUID	A universally unique identifier.	[2]
VARCHAR(*n*)	CHARACTER VARYING(*n*)	A varying-length character string.	VARCHAR(*n*)
[2]	XML	XML data.	[2]

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The PostgreSQL data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Reading from Salesforce

The following table lists the data type support for reading from Salesforce. SAS/ACCESS software uses the Salesforce SOAP API to read from Salesforce. The Salesforce SOAP API returns primitive types.

SAS supports the Winter '19 release of Salesforce and later.

The calculated type and compound types are not supported. Examples of compound types are address and location.

For more information, see documentation for the Salesforce SOAP API.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about Salesforce support in CAS, see the SAS Viya documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A7.22 *Data Type Mapping When Reading from Salesforce*

Salesforce Field or Primitive Type	Description	Data Type Returned by FedSQL
anyType	A polymorphic data type that can be used differently throughout the interface based on context.	VARCHAR(*n*)
base64	A Base-64 binary string, including attachment records, document records, and scontrol records. The Body/Binary field contains data. The BodyLength field defines length of data. [1]	VARCHAR(*n*)
boolean	A Boolean value.	TINYINT
byte	A set of bits.	TINYINT
combobox	A set of enumerated values that allows the user to define a value that is not in the list.	VARCHAR(*n*)
currency	A currency value.	DOUBLE
DataCategoryGroupReference	A reference to a data category group or category unique name on Article and Question objects.	VARCHAR(*n*)
date	A date value.	DATE[2]

Salesforce Field or Primitive Type	Description	Data Type Returned by FedSQL
dateTime	A timestamp.	TIMESTAMP[2]
double	A signed, double precision, floating-point number.	DOUBLE[3]
email	An email address.	VARCHAR(*n*)
encryptedstring	An encrypted text string that contain a combination of letters, numbers, and symbols that are stored in encrypted form. The strings have a maximum length of 175 characters.	VARCHAR(*n*)
ID	The primary key value for an object.	VARCHAR(*n*)
int	A regular signed, exact whole number whose size varies depending on field settings.	BIGINT
JunctionIdList	A string array of referenced IDs values that represent the many-to-many relationship of an underlying junction entity.	VARCHAR(*n*)
masterrecord	Identifier of a record that is merged and the original records are deleted.	VARCHAR(*n*)
multipicklist	A set of enumerated values from which multiple values can be selected. The selected values are returned as a string of semicolon-separated values.	VARCHAR(*n*)
percent	A percentage value.	DOUBLE
phone	A phone number.	CHAR(40)
picklist	A set of enumerated values from which one value can be selected.	VARCHAR(*n*)
reference	A cross-reference for a different object. Analogous to a foreign key in SQL.	VARCHAR(*n*)
String	A text string whose maximum size varies depending on field settings.	CHAR(*n*)
textarea	A string that is displayed as a multiline text field.	VARCHAR(*n*)
Time	A time value.	TIME[2]

Salesforce Field or Primitive Type	Description	Data Type Returned by FedSQL
url	A Uniform Resource Locator (URL) value.	VARCHAR(*n*)

1 base64 fields remain in base64 format as a VARCHAR type.

2 The earliest valid date is 1700-01-01T00:00:00Z GMT, or just after midnight on January 1, 1700. The latest valid date is 4000-12-31T00:00:00Z GMT, or just after midnight on December 31, 4000. These values are offset by time zone. For example, in the Pacific time zone, the earliest valid date is 1699-12-31T16:00:00, or 4:00 PM on December 31, 1699.

3 double columns that match the precision and scale described for the numeric types in "Data Types for Writing to Salesforce" on page 1149 are mapped as shown in that table. For example, a double column that has a precision of 10 and scale of 0 is mapped to INTEGER.

Data Types for Writing to Salesforce

The following table lists the data type support for writing to Salesforce. SAS/ACCESS software uses the Salesforce Metadata API to write to Salesforce. The Salesforce Metadata API uses Custom Field types.

SAS supports the Winter '19 release of Salesforce and later.

The BINARY, FLOAT, and VARBINARY data types are not supported for data definition.

For more information, see documentation for the Salesforce Metadata API.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about Salesforce support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A7.23 *Data Type Mapping When Writing to Salesforce*

Data Type Definition Keyword	Salesforce Custom Field Type	Description	Data Type Returned by CREATE TABLE with AS Expression
BIGINT	Number	A number with precision 18, scale 0.	BIGINT
CHAR(*n*)[1]	Text	A character string that is less than or equal to 255 characters.	CHAR(*n*)
DATE	Date	A date value.[2]	DATE
DECIMAL| NUMERIC(*p*,*s*)	Number	A number with precision and scale.	DOUBLE

Data Type Definition Keyword	Salesforce Custom Field Type	Description	Data Type Returned by CREATE TABLE with AS Expression
DOUBLE	Number	A number with precision 18, scale 4.	DOUBLE
INTEGER	Number	A number with precision 10, scale 0.	INTEGER
NCHAR(*n*)	Text	A Unicode character string of fixed length.	CHAR(*n*)
NVARCHAR(*n*)	Text	A Unicode character string of varying length.	VARCHAR(*n*)
REAL	Number	A number with precision 18, scale 4.	DOUBLE
SMALLINT	Number	A number with precision 5, scale 0.	SMALLINT
TIME	Time	A time value. [2]	TIME
TIMESTAMP	Datetime	A timestamp. [2]	TIMESTAMP
TINYINT	Number	A number with precision 3, scale 0.	TINYINT
VARCHAR(*n*)[3]	LongTextArea	A character string that is greater than 255 characters and less than 32K characters.	VARCHAR(*n*)

1 CHAR(*n*) values where *n* is greater than 255 characters are created as VARCHAR(*n*).

2 The earliest valid date is 1700-01-01T00:00:00Z GMT, or just after midnight on January 1, 1700. The latest valid date is 4000-12-31T00:00:00Z GMT, or just after midnight on December 31, 4000. These values are offset by time zone. For example, in the Pacific time zone, the earliest valid date is 1699-12-31T16:00:00, or 4:00 PM on December 31, 1699.

3 VARCHAR(*n*) values where *n* is less than or equal to 255 characters are created as CHAR(*n*).

Data Types for SAP

The following table lists the data type support for an SAP system.

For an SAP system, no data types are supported for column definition. Native ABAP SAP data types are mapped to similar data types for data retrieval only.

For data source-specific information about the ABAP SAP data types, see the SAP system database documentation.

Note: This data source is not supported on the CAS server.

Table A7.24 Mapping of SAP Data Types to FedSQL Data Types

ABAP SAP Data Type	Description	FedSQL Data Type Used For Data Retrieval
ACCP	Posting period.	CHAR(*n*) for non-Unicode SAP system; NCHAR(*n*) for Unicode SAP system
CHAR	A fixed-length character string.	CHAR(*n*) for non-Unicode SAP system; NCHAR(*n*) for Unicode SAP system
CLNT	A client value.	CHAR(*n*) for non-Unicode SAP system; NCHAR(*n*) for Unicode SAP system
CUKY	A currency key. Fields of this type are referenced by fields of type CURR.	CHAR(*n*) for non-Unicode SAP system; NCHAR(*n*) for Unicode SAP system
CURR	A currency value. Corresponds to the DEC field. Field refers to a field of type CUKY.	CHAR(*n*)
DATS	A date value.	DATE
DEC	A signed, fixed-point decimal number.	CHAR(*n*)
FLTP	A floating-point number.	DOUBLE
INT1	A very small signed, exact whole number.	TINYINT
INT2	A small signed, exact whole number.	SMALLINT
INT4	A regular signed, exact whole number.	INTEGER
LANG	A language key, which has its own field format for special functions. The conversion exit ISOLA converts the value to be displayed to that of the database and the opposite is true.	CHAR(*n*) for non-Unicode SAP system; NCHAR(*n*) for Unicode SAP system
LCHR	A fixed-length character string.	VARCHAR(*n*) for non-Unicode SAP system; NVARCHAR(*n*) for Unicode SAP system
LRAW	An uninterpreted varying-length byte string.	VARBINARY(*n*)

ABAP SAP Data Type	Description	FedSQL Data Type Used For Data Retrieval
NUMC	Text string.	CHAR(*n*) for non-Unicode SAP system; NCHAR(*n*) for Unicode SAP system
PREC	The precision of a QUAN field.	CHAR(*n*)
QUAN	A quantity that corresponds to the DEC field.	CHAR(*n*)
RAW	An uninterpreted byte string.	BINARY
TIMS	A time value.	TIME(*p*)
UNIT	Units key and referenced by a QUAN data type.	CHAR(*n*) for non-Unicode SAP system; NCHAR(*n*) for Unicode SAP system
VARC	A varying-length character string data. As of SAP release 3.0, creating fields of this data type is no longer supported. Existing fields with this data type can be used, except in a WHERE condition in the SELECT statement.	VARCHAR(*n*) for non-Unicode SAS system; NVARCHAR(*n*) for Unicode SAP system

Data Types for SAP HANA

The following table lists the data type support for an SAP HANA database.

For data source-specific information about the SAP HANA data types, see the SAP HANA database documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about SAP HANA support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A7.25 *Mapping of FedSQL Data Types to Data Types Used by SAP HANA*

Data Type Definition Keyword [1]	SAP HANA Data Type	Description	Data Type Returned
[2]	ALPHANUM(*n*)	A varying-length character string.	NVARCHAR(*n*)

Data Type Definition Keyword [1]	SAP HANA Data Type	Description	Data Type Returned
BIGINT	BIGINT	64-bit integer.	BIGINT
BINARY(*n*)	BINARY(*n*)	A fixed-length binary data.	BINARY(*n*)
[2]	BLOB	A varying-length binary large object string.	[2]
CHAR(*n*)	CHAR(*n*)	A varying-length character string.	CHAR(*n*)
[2]	CLOB	A varying-length character large object string.	[2]
DATE	DATE	Year, month, and day values.	DATE
DECIMAL| NUMERIC(p,*s*)	DECIMAL(*p*,*s*)	A signed, exact, fixed-point decimal number.	DECIMAL(*p*,*s*)[3]
DOUBLE	DOUBLE	Double-precision floating-point number.	DOUBLE
FLOAT(*p*)	FLOAT(*n*)	A floating-point number.	FLOAT(*p*)
INTEGER	INTEGER	32-bit integer.	INTEGER
NCHAR(*n*)	NCHAR(*n*)	A fixed-length Unicode character string.	NCHAR(*n*)
[2]	NCLOB	A fixed-length character large object string.	[2]
NVARCHAR(*n*)	NVARCHAR(*n*)	A varying-length Unicode character large object string.	NVARCHAR(*n*)
REAL	REAL	A floating-point number.	REAL
[2]	SECONDATE	A date and time value.	[2]
[2]	SHORTTEXT(*n*)	A varying-length character string.	NVARCHAR(*n*)
[2]	SMALLDECIMAL(*p*,*s*)	A floating-point decimal number.	DECIMAL(*p*,*s*)[3]
SMALLINT	SMALLINT	16-bit integer.	SMALLINT
[2]	TEXT	A varying-length Unicode character large object string.	[2]
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)

Data Type Definition Keyword [1]	SAP HANA Data Type	Description	Data Type Returned
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(p)
TINYINT	TINYINT	An unsigned eight-bit integer.	TINYINT
VARBINARY(*n*)	VARBINARY(*n*)	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The SAP HANA data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for SAP IQ

The following table lists the data type support for an SAP IQ database.

For data source-specific information about the SAP IQ database data types, see the SAP IQ database documentation.

Note: This data source is not supported on the CAS server.

Table A7.26 *Mapping of FedSQL Data Types to Data Types Used by SAP IQ*

Data Type Definition Keyword[1]	SAP IQ Data Type	Description	Data Type Returned
BIGINT	BIGINT	64-bit integer.	BIGINT
BINARY(*n*)	BINARY(*n*)	A fixed-length binary string.	BINARY(*n*)
[3]	BIT	Integer that stores only the values 0 or 1.	[3]
CHAR(*n*)	CHAR(*n*)	A varying-length character string.	VARCHAR(*n*)
DATE	DATE	A date value.	DATE

Data Type Definition Keyword[1]	SAP IQ Data Type	Description	Data Type Returned
DECIMAL| NUMERIC(*p*,*s*)	DECIMAL(*p*,*s*)	A signed, exact, fixed-point decimal number.	DECIMAL(*p*,*s*)[4]
DOUBLE	DOUBLE	Double-precision floating-point number.	DOUBLE
FLOAT(*p*)	FLOAT(*p*)	A floating-point number.	FLOAT(*p*)
INTEGER	INTEGER	32-bit integer.	INTEGER
[2]	LONG BINARY	A varying-length binary string.	VARBINARY(*n*)
REAL	REAL	A floating-point number.	REAL
SMALLINT	SMALLINT	16-bit integer.	SMALLINT
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
TINYINT	TINYINT	An unsigned eight-bit integer.	TINYINT
VARBINARY(*n*)	VARBINARY(*n*)	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)	CHAR(*n*)	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The SAP IQ data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The SAP IQ data type cannot be defined or retrieved.

4 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Snowflake

The following table lists the data type support for Snowflake. SAS support for Snowflake is based on version 2.19.2 or later of the Snowflake ODBC driver.

The FedSQL NCHAR, NVARCHAR, and TINYINT data types are not supported for data definition.

For data source-specific information about the Snowflake data types, see the Snowflake documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about Snowflake support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A7.27 *Mapping of FedSQL Data Types to Data Types Used by Snowflake*

Data Type Definition Keyword	Snowflake Data Type	Description	Data Type Returned
[1]	ARRAY	Dense or sparse arrays of arbitrary size, and values have VARIANT type.	VARCHAR
BIGINT	BIGINT	64-bit integer.	BIGINT
BINARY(*n*)	BINARY(*n*)	A fixed-length binary string.	BINARY
[1]	BOOLEAN	A true, false, or unknown (null) value.	BINARY
CHAR(*n*)	VARCHAR(1)	A fixed-length character string.	VARCHAR
DATE	DATE	A date value.	DATE
DECIMAL| NUMERIC(*p*,*s*)	NUMBER(*p*,*s*)	A signed, exact, fixed-point decimal number.	DECIMAL(*p*,*s*)[2]
DOUBLE	FLOAT	Double-precision floating-point number.	DOUBLE
FLOAT(*p*)	FLOAT, FLOAT4, FLOAT8	A floating-point number.	FLOAT(*p*)
INTEGER	NUMBER without precision and scale	32-bit integer.	INTEGER
[1]	OBJECT	a collection of key-value pairs, where the key is a non-empty string, and the value is a value of the VARIANT type.	VARCHAR
REAL	FLOAT	A floating-point number.	FLOAT
SMALLINT	NUMBER without precision and scale	16-bit integer.	SMALLINT

Data Type Definition Keyword	Snowflake Data Type	Description	Data Type Returned
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP, TIMESTAMP_LTZ, TIMESTAMP_NTZ, TIMESTAMP_TZ	A date and time value.	TIMESTAMP(*p*)
VARBINARY(*n*)	VARBINARY	A varying-length binary string.	BINARY
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string. The default (and maximum) length is 16,777,216.	VARCHAR
[1]	VARIANT	A universal data type that can store values of any other Snowflake data type, including OBJECT and ARRAY, up to 16 MB.	VARCHAR

1 The Snowflake data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

2 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Spark

The following table lists the data type support for Apache Spark.

The NCHAR and NVARCHAR data types are not available for data definition.

For data source-specific information about Spark SQL data types, see the Spark SQL documentation.

Note: This data source is not supported on the CAS server.

Table A7.28 *Mapping of FedSQL Data Types to Data Types Used by Apache Spark*

Data Type Definition Keyword	Spark Data Type	Description	Data Type Returned
[1]	ARRAY<*data-type*>	An array of integers (indexable lists).	STRING[5, 7]

Data Type Definition Keyword	Spark Data Type	Description	Data Type Returned
BIGINT	BIGINT	A signed eight-byte integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.	BIGINT
BINARY(*n*)	BINARY	A varying-length binary string up to 32K.	BINARY
[1]	BOOLEAN	A textual true or false value.	TINYINT
CHAR(*n*)	CHAR(*n*)[9]	A character string up to 255 characters. [3]	CHAR
DATE	DATE[2]	An ANSI SQL date value.	DATE
DECIMAL/ NUMERIC(*p*,*s*)	DECIMAL	A fixed-point decimal number, with 38 digits precision.	DECIMAL(*p*,*s*)[4]
DOUBLE	DOUBLE	An eight-byte, double-precision floating-point number.	DOUBLE
FLOAT(*p*)	FLOAT	A four-byte, single-precision floating-point number.	DOUBLE
INTEGER	INTEGER	A signed four-byte integer.	INTEGER
[1]	MAP<*primitive-type*, *data-type*>	An associative array of key-value pairs.	STRING[5, 7]
REAL	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
SMALLINT	SMALLINT	A signed two-byte integer, from -32,768 to 32,767.	SMALLINT
[1]	STRING	A varying-length character string.	VARCHAR(*n*)[5, 6]

Data Type Definition Keyword	Spark Data Type	Description	Data Type Returned
[1]	STRUCT<*col-name*: *data-type*>	A structure with established column elements and data types. Column elements and data types are mapped using a double-dot (:) notation.	STRING[5, 7]
TIME(*p*)	[8]	A time value.	STRING
TIMESTAMP(*p*)	TIMESTAMP	A UNIX timestamp with optional nanosecond precision.	TIMESTAMP[(*p*)]
TINYINT	TINYINT	A signed one-byte integer, from -128 to 127.	TINYINT
VARBINARY	BINARY	A varying-length binary string up to 32K.	BINARY
VARCHAR(*n*)	VARCHAR(*n*)[9]	A varying-length character string.	VARCHAR(*n*)[5]

1 The Spark data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.
2 The supported date values are between January 1, 0001 and December 31, 9999. Date values later than 9999 are read back as null values.
3 If you specify CHAR with a value greater than 255 characters, the column is created as VARCHAR(*n*) instead.
4 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.
5 The maximum length of VARCHAR(*n*) and the Hive complex types is determined by the DBMAX_TEXT= data source connection option.
6 SASFMT TableProperties are applied when reading STRING columns.
7 The complex types ARRAY, MAP, STRUCT are read as their STRING representation of the underlying complex type. ARRAY values are read back within brackets, for example: [1, 2, 4]. STRUCT and MAP values are read back within braces, for example: {"firstname":"robert","nickname":"bob"}.
8 Spark does not support the TIME(*p*) data type. When data is read from Spark, STRING columns that have SASFMT TableProperties defined that specify the SAS TIME8. format are converted to the TIME(*p*) data type. When the TIME type is used for data definition, it is mapped to a STRING column with SASFMT TableProperties. Fractional seconds are not preserved. For more information about SASFMT TableProperties, see “SAS Table Properties for Spark and Hadoop” in *SAS/ACCESS for Relational Databases: Reference*.
9 In Apache Spark 2.0.0, the maximum number of columns allowed in a Spark table might be limited by https://issues.apache.org/jira/browse/SPARK-18016 .

Data Types for Teradata

The following table lists the data type support for a Teradata database.

The NCHAR, NVARCHAR, and REAL data types are not supported for data type definition.

For data source-specific information about the Teradata data types, see the Teradata database documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about Teradata support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A7.29 *Mapping of FedSQL Data Types to Data Types Used by Teradata*

Data Type Definition Keyword[1]	Teradata Data Type	Description	Data Type Returned
BIGINT	BIGINT	A large signed, exact whole number.	BIGINT
BINARY(*n*)	BYTE(*n*)	A fixed-length binary string	BINARY(*n*)
[2]	BLOB	A large Binary Object string.	[2]
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
[2]	CLOB	A large Character Object string.	[2]
DATE	DATE	A date value.	DATE
DECIMAL(*p*,*s*)	DECIMAL(*p*,*s*)	A signed, fixed-point decimal number.	DECIMAL(*p*,*s*)[3]
NUMERIC(*p*,*s*)	NUMBER(*p*,*s*)	A signed, varying-length decimal number.	NUMERIC(*p*,*s*)[3]
DOUBLE	FLOAT	A signed, double precision, floating-point number.	DOUBLE
FLOAT(*p*)	FLOAT(*p*)	A signed, double precision, floating-point number.	FLOAT(*p*)
INTEGER	INTEGER	A regular signed, exact whole number.	INTEGER
[2]	LONG VARCHAR	A varying-length character string.	[2]
SMALLINT	SMALLINT	A small signed, exact whole number	SMALLINT
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)

Data Type Definition Keyword[1]	Teradata Data Type	Description	Data Type Returned
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
TINYINT	BYTEINT	A very small signed, exact whole number.	TINYINT
VARBINARY(*n*)	VARBYTE(*n*)	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The Teradata data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The DECIMAL and NUMERIC data types are supported in requests that can be fully passed down to the data source. DECIMAL and NUMERIC columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Vertica

The following table lists the data type support for a Vertica database.

The NCHAR and NVARCHAR data types are not supported for data definition.

For data source-specific information about Vertica data types, see the Vertica database documentation.

Note: This data source is not supported on the CAS server.

Table A7.30 *Mapping of FedSQL Data Types to Data Types Used by Vertica*

Data Type Definition Keyword [1]	Vertica SQL Identifier	Description	Data Type Returned
BIGINT	BIGINT	A signed 64-bit integer, requiring 8 bytes of storage.	BIGINT
BINARY(*n*)	BINARY(*n*)	A fixed-length binary string.	BINARY(*n*)
[2]	BOOLEAN	A logical Boolean (true/false) value.	[2]

Data Type Definition Keyword [1]	Vertica SQL Identifier	Description	Data Type Returned
[2]	BYTEA	A varying-length binary string.	[2]
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
DATE	DATE	A date value.	DATE
[2]	DATETIME	A date and time value with or without time zone.	TIMESTAMP(*p*)
DECIMAL| NUMERIC(*p*,*s*)	DECIMAL| NUMERIC(*p*,*s*)	A signed, fixed precision and scale numbers. Default precision and scale: 37, 15.	DECIMAL| NUMERIC(*p*,*s*)[3]
DOUBLE	DOUBLE PRECISION	A signed 64-bit IEEE floating point number, requiring 8 bytes of storage	DOUBLE
FLOAT(*p*)	FLOAT(*n*), FLOAT8	A signed 64-bit IEEE floating point number, requiring 8 bytes of storage	DOUBLE
INTEGER	INTEGER	A signed 64-bit integer, requiring 8 bytes of storage.	BIGINT
[2]	INTERVAL	Time span.	[2]
[2]	INTERVAL DAY TO SECOND	Time span.	[2]
[2]	INTERVAL YEAR TO MONTH	Time span.	[2]
[2]	LONG VARCHAR	Varying-length, raw-byte data, such as spatial data.	[2]
[2]	LONG BINARY	Long varying length binary string.	[2]
[2]	MONEY(*p*,*s*)	A currency value of signed, fixed precision and scale. Default	[2]

Data Type Definition Keyword [1]	Vertica SQL Identifier	Description	Data Type Returned
		precision and scale: 18, 4.	
[2]	NUMBER(*p*,*s*)	A signed, fixed precision and scale numbers. Default precision and scale: 38,0.	[2]
[2]	RAW	A varying-length binary string.	[2]
REAL	REAL	A signed 64-bit IEEE floating point number, requiring 8 bytes of storage.	DOUBLE
[2]	SMALLDATETIME	A date and time value with or without time zone.	TIMESTAMP(*p*)
SMALLINT	SMALLINT	A signed 64-bit integer, requiring 8 bytes of storage.	BIGINT
TIME(*p*)	TIME(*p*)	A time value without time zone.	TIME(*p*)
[2]	TIMETZ	A time value with time zone.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP	A date and time without time zone.	TIMESTAMP(*p*)
[2]	TIMESTAMPTZ	A date and time with time zone	TIMESTAMP(*p*)
TINYINT	TINYINT	A signed 64-bit integer, requiring 8 bytes of storage.	BIGINT
VARBINARY(*n*)	VARBINARY	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)	VARCHAR(*n*), TEXT	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined.

The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The Vertica data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Yellowbrick

The following table lists the data type support for a Yellowbrick database.

The BINARY, NCHAR, NVARCHAR, TINYINT, and VARBINARY data types are not supported for data type definition.

Note: This data source is not supported on the CAS server.

For data source-specific information about Yellowbrick data types, see the Yellowbrick database documentation.

Table A7.31 *Mapping of FedSQL Data Types to Data Types Used by Yellowbrick*

Data Type Definition Keyword[1]	Yellowbrick Data Type	Description	Data Type Returned
BIGINT	BIGINT	A large signed, exact whole number or a signed eight-byte integer.	BIGINT
[2]	BOOLEAN	A logical Boolean (true/false) value.	[2]
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
DATE	DATE	A date value.	DATE
DECIMAL| NUMERIC(*p*,*s*)	NUMERIC(*p*,*s*)	A signed, fixed-point decimal number.	DECIMAL| NUMERIC(*p*,*s*)[3]
DOUBLE	DOUBLE PRECISION	A signed, double precision, floating-point number.	DOUBLE
FLOAT(*p*)	DOUBLE PRECISION, FLOAT, FLOAT8,	A signed, single precision or double precision, floating-point number.	FLOAT

Data Type Definition Keyword[1]	Yellowbrick Data Type	Description	Data Type Returned
INTEGER	INTEGER	A regular signed, exact whole number.	INTEGER
[2]	MACADDR, MACADDR8	A Media Access Control address.	[2]
REAL	REAL, FLOAT4	A signed, single precision floating-point number.	REAL
SMALLINT	SMALLINT	A small signed, exact whole number.	SMALLINT
TIME(*p*)	TIME	A time value.	TIME
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
[2]	UUID	A universally unique identifier.	[2]
VARCHAR(*n*)	CHARACTER VARYING(*n*)	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The Yellowbrick data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Appendix 3

Using FedSQL and DS2

Dynamically Executing FedSQL Statements from DS2

Note: With the exception of the SET statement in a DS2 action or PROC DS2, using a FedSQL query within your DS2 program is not currently supported on the CAS server.

You can embed and execute FedSQL statements from within your DS2 programs. You can use FedSQL with DS2 in the following instances:

- You can invoke a DS2 package method expression as a function in a FedSQL SELECT statement.

  For more information, see "Using DS2 Packages in Expressions" on page 216.

- You can use the SQLSTMT package to generate, prepare, and execute FedSQL statements to create, select, modify, insert, or delete rows from a table at run time.

  The SQLSTMT package is intended for use with FedSQL statements that are executed multiple times, statements with parameters, or statements that generate a result set. For more information, see "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*.

- You can also use the SQLEXEC function to generate, prepare, and execute FedSQL statements to create, select, modify, insert, or delete rows from a table at run time.

  The SQLEXEC function is intended for use with FedSQL statements that are executed only one time, do not have parameters, and do not produce a result set.

For more information, see the "SQLEXEC Function" in *SAS DS2 Language Reference*.

- You can load data into a hash instance at run time by using a FedSQL SELECT statement in the DECLARE PACKAGE statement or the DATASET method.

  For more information, see "Using a FedSQL Query with a Hash Instance to Get Rows Dynamically at Run Time" in *SAS DS2 Programmer's Guide* the "DATASET Method" in *SAS DS2 Language Reference* and the "DECLARE PACKAGE Statement: Hash Package" in *SAS DS2 Language Reference*.

- You can use the SET statement to read in data by using a FedSQL SELECT statement.

  For more information, see "SET Statement with Embedded FedSQL" in *SAS DS2 Programmer's Guide* and the "SET Statement" in *SAS DS2 Language Reference*.

Appendix 4

Tables Used in Examples

Introduction

This section provides the code to create the tables used in the examples provided in this book. It also provides output listings that show the content of the tables.

AFewWords

Code

```
create table afewwords(Word1 char(10), Word2 char(10));
insert into afewwords values ('*some/', 'WHERE');
insert into afewwords values ('*every*', 'THING');
insert into afewwords values ('*no*', 'BODY');

select * from afewwords;
```

Content

Output A9.1 *Content of Table AfewWords*

WORD1	WORD2
*some/	WHERE
every	THING
no	BODY

Customers

Code

```
create table customers
     ( custid double primary key,
     name char(16),
     address char(64),
     city char(16),
     state char(2),
     country char(16),
     phone char(16),
     initorder date
     );

insert into customers values (1,'Peter Frank', '300 Rock Lane',
'Boulder', 'CO',
'United States', '3039564321', date '2012-01-14');
insert into customers values (2,'Jim Stewart', '1500 Lapis Lane',
'Little Rock',
'AR', 'United States', '8705553978', date '2012-03-20');
insert into customers values (3,'Janet Chien', '75 Jujitsu',
'Nagasaki', '',
'Japan', '01181956879932', date '2012-06-07');
insert into customers values (4,'Qing Ziao', '10111 Karaje', 'Tokyo',
'',
'Japan', '0118136774351', date '2012-10-12');
insert into customers values (5,'Humberto Sertu', '876 Avenida
Blanca', 'Buenos Aires',
'', 'Argentina','01154118435029', date '2012-12-15');
```

```
select * from customers;
```

Content

Output A9.2 *Content of Table Customers*

CUSTID	NAME	ADDRESS	CITY	STATE	COUNTRY	PHONE	INITORDER
1	Peter Frank	300 Rock Lane	Boulder	CO	United States	3039564321	14JAN2012
2	Jim Stewart	1500 Lapis Lane	Little Rock	AR	United States	8705553978	20MAR2012
3	Janet Chien	75 Jujitsu	Nagasaki		Japan	01181956879932	07JUN2012
4	Qing Ziao	10111 Karaje	Tokyo		Japan	0118136774351	12OCT2012
5	Humberto Sertu	876 Avenida Blanca	Buenos Aires		Argentina	01154118435029	15DEC2012

CustonLine

Code

```
create table custonline(custnum varchar(15) having label 'Customer
Number',
       begintime timestamp having label 'Begin Time',
       endtime timestamp having label 'End Time' format datetime23.3);

insert into custonline values ('US-C-37533944', timestamp'2013-09-01
10:00:00',
timestamp'2013-09-01 10:05:01.253');
insert into custonline values ('GB-W-33944332', timestamp'2013-10-02
22:15:33',
timestamp'2013-10-02 22:21:09.421');
insert into custonline values ('SP-M-29443992', timestamp'2013-10-15
18:44:25',
timestamp'2013-10-15 19:04:55.746');
insert into custonline values ('US-A-37144324', timestamp'2013-11-01
12:03:59',
timestamp'2013-11-01 12:25:09.398');
insert into custonline values ('FR-P-98384488', timestamp'2013-12-01
12:15:34',
timestamp'2013-12-01 12:47:45.221');
insert into custonline values ('GB-L-24995559', timestamp'2013-01-02
15:43:24',
timestamp'2013-01-02 16:06:15.766');
insert into custonline values ('FR-L-42339887', timestamp'2013-01-16
14:55:00',
```

```
timestamp'2013-01-16 15:05:56.288');
insert into custonline values ('GB-P-87559899', timestamp'2013-02-01
11:02:44',
timestamp'2013-02-01 11:15:33.955');
insert into custonline values ('SP-N-44333958', timestamp'2013-03-01
10:14:33',
timestamp'2013-03-01 10:35:27.908');
insert into custonline values ('GB-R-24994990', timestamp'2013-03-15
09:06:00',
timestamp'2013-03-15 09:06:20.475');

select * from custonline;
```

Content

Output A9.3 *Content of Table CustonLine*

Customer Number	Begin Time	End Time
US-C-37533944	01SEP13:10:00:00.00	01SEP2013:10:05:01.253
GB-W-33944332	02OCT13:22:15:33.00	02OCT2013:22:21:09.421
SP-M-29443992	15OCT13:18:44:25.00	15OCT2013:19:04:55.746
US-A-37144324	01NOV13:12:03:59.00	01NOV2013:12:25:09.398
FR-P-98384488	01DEC13:12:15:34.00	01DEC2013:12:47:45.221
GB-L-24995559	02JAN13:15:43:24.00	02JAN2013:16:06:15.766
FR-L-42339887	16JAN13:14:55:00.00	16JAN2013:15:05:56.288
GB-P-87559899	01FEB13:11:02:44.00	01FEB2013:11:15:33.955
SP-N-44333958	01MAR13:10:14:33.00	01MAR2013:10:35:27.908
GB-R-24994990	15MAR13:09:06:00.00	15MAR2013:09:06:20.475

Densities

Code

```
proc fedsql;
```

```
create table densities(Name char(20),
     Population double having format comma12.,
            SquareMiles double having format comma12.,
            Density double);

insert into densities values ('Afghanistan', 17070323, 251825, 67.79);
insert into densities values ('Albania', 3407400, 11100, 306.97);
insert into densities values ('Algeria', 28171132, 919595, 30.63);
insert into densities values ('Andorra', 64634, 200, 323.17);
insert into densities values ('Angola', 9901050, 481300, 20.57);
insert into densities values ('Antigua and Bar', 65644, 171, 383.88);
insert into densities values ('Argentina', 34248705, 1073518, 31.90);
insert into densities values ('Armenia', 3556864, 11500, 309.29);
insert into densities values ('Australia', 18255944, 2966200, 6.15);
insert into densities values ('Austria', 8033746, 32400, 247.96);

select * from densities;
```

Content

Output A9.4 *Content of Table Densities*

NAME	POPULATION	SQUAREMILES	DENSITY
Afghanistan	17,070,323	251,825	67.79
Albania	3,407,400	11,100	306.97
Algeria	28,171,132	919,595	30.63
Andorra	64,634	200	323.17
Angola	9,901,050	481,300	20.57
Antigua and Bar	65,644	171	383.88
Argentina	34,248,705	1,073,518	31.9
Armenia	3,556,864	11,500	309.29
Australia	18,255,944	2,966,200	6.15
Austria	8,033,746	32,400	247.96

Depts

Code

```
create table depts(deptno double,
     deptname char(30) not null,
         manager char(30)
     );
```

```
insert into depts values (10, 'Sales', 'Jim Barnes');
insert into depts values (20, 'Research', 'Clifford James');
insert into depts values (30, 'Accounting', 'Barbara Sandman');
insert into depts values (40, 'Operations', 'William Baylor');

select * from depts;
```

Content

Output A9.5 *Content of Table Depts*

DEPTNO	DEPTNAME	MANAGER
10	Sales	Jim Barnes
20	Research	Clifford James
30	Accounting	Barbara Sandman
40	Operations	William Baylor

Employees

Code

```
create table employees
     (empid double,
      dept double,
      emp_name char(30),
          pos char(50),
      hire_date date
         );

insert into employees values (1, 10, 'Jim Barnes', 'Manager',
date '2000-11-26');
insert into employees values (2, 20, 'Clifford James', 'Manager',
date '2000-11-26');
insert into employees values (3, 30, 'Barbara Sandman', 'Manager',
date '2000-11-26');
insert into employees values (4, 40, 'William Baylor', 'Manager',
date '2000-11-26');
insert into employees values (5, 20, 'Greg Welty', 'Developer',
```

```
date '2004-11-26');
insert into employees values (6, 20, 'Penny Jackson', 'Developer',
date '2004-11-26');
insert into employees values (7, 10, 'Edward Murray', 'Sales
Associate',
date '2004-11-26');
insert into employees values (8, 10, 'Ronald Thomas', 'Sales
Associate',
date '2004-11-26');
insert into employees values (9, 30, 'Elsie Marks', 'Executive
Assistant',
date '2005-02-11');
insert into employees values (10, 40, 'Bruno Kramer', 'Grounds support
technician',
date '2005-11-02');

select * from employees;
```

Content

Output A9.6 *Content of Employees Table*

EMPID	DEPT	EMP_NAME	POS	HIRE_DATE
5	20	Greg Welty	Developer	26NOV2004
6	20	Penny Jackson	Developer	26NOV2004
7	10	Edward Murray	Sales Associate	26NOV2004
8	10	Ronald Thomas	Sales Associate	26NOV2004
9	30	Elsie Marks	Executive Assistant	11FEB2005
10	40	Bruno Kramer	Grounds support technician	02NOV2005

GrainProducts

Code

```
create table grainproducts(prodid double primary key, product char(8));

 insert into grainproducts values (1424, 'Rice');
 insert into grainproducts values (3421, 'Corn');
```

```
 insert into grainproducts values (3234, 'Wheat');
 insert into grainproducts values (3485, 'Oat');

select * from grainproducts;
```

Content

Output A9.7 *Content of GrainProducts Table*

PRODID	PRODUCT
1424	Rice
3421	Corn
3234	Wheat
3485	Oat

Integers

Code

```
create table integers
     (bi       bigint having format comma25.0,
      ii       integer,
      si       smallint,
      ti       tinyint);

   insert into integers
values(9223372036854775800,2147483647,32767,127);

      select * from integers;
```

Content

Output A9.8 *Content of Integers Table*

BI	II	SI	TI
9,223,372,036,854,775,808	2.1475E9	32767	127

Numbers1

Code

```
create table numbers1(x char(5), y double, z double);
insert into numbers1 values ('one', 1, 10);
insert into numbers1 values ('three', 3, 30);
insert into numbers1 values ('four', 4, 40);
insert into numbers1 values ('four', 4, 40);
insert into numbers1 values ('five', 5, 50);
insert into numbers1 values('five', 5, 50);

select * from numbers1;
```

Content

Output A9.9 *Content of Table Numbers1*

X	Y	Z
one	1	10
three	3	30
four	4	40
four	4	40
five	5	50
five	5	50

Numbers2

Code

```
create table numbers2(x varchar(5), y double, z double);
insert into numbers2 values ('four', 4, 40);
insert into numbers2 values ('five', 5, 50);
insert into numbers2 values('five', 5, 50);
insert into numbers2 values ('one', 1, 10);
insert into numbers2 values ('two', 2, 20);

select * from numbers2;
```

Content

Output A9.10 *Content of Table Numbers2*

X	Y	Z
four	4	40
five	5	50
five	5	50
one	1	10
two	2	20

Numbers3

Code

```
create table numbers3(z double, y double, x varchar(6));
insert into numbers3 values (40, 4, 'four');
insert into numbers3 values(60, 6, 'six');
insert into numbers3 values(70, 7, 'seven');
```

```
select * from numbers3;
```

Content

Output A9.11 *Content of Table Numbers3*

Z	Y	X
40	4	four
60	6	six
70	7	seven

Products

Code

```
create table products(prodid double primary key, product char(20));

   insert into products values (3234, 'Rice');
   insert into products values (1424, 'Corn');
   insert into products values (3421, 'Wheat');
   insert into products values (3422, 'Oat');
   insert into products values (3975, 'Barley');

select * from products;
```

Content

Output A9.12 *Content of Products Table*

PRODID	PRODUCT
3234	Rice
1424	Corn
3421	Wheat
3422	Oat
3975	Barley

Sales

Code

```
create table sales
   ( prodid double,
     custid double primary key,
     totals double having format dollar12.,
     country char(30) not null
   );

insert into sales values (3234, 1, 189400, 'United States');
insert into sales values (1424, 3, 555789, 'Japan');
insert into sales values (3421, 4, 781183, 'Japan');
insert into sales values (3421, 2, 2789654, 'United States');
insert into sales values (3975, 5, 899453, 'Argentina');

select * from sales;
```

Content

Output A9.13 *Content of Table Sales*

PRODID	CUSTID	TOTALS	COUNTRY
3234	1	$189,400	United States
1424	3	$555,789	Japan
3421	4	$781,183	Japan
3421	2	$2,789,654	United States
3975	5	$899,453	Argentina

WorldCityCoords

Code

```
create table worldcitycoords(city char(20),
     country char(20),
     latitude double,
     longitude double);

insert into worldcitycoords values ('Algiers', 'Algeria', 37, 3);
insert into worldcitycoords values ('Amsterdam', 'Netherlands', 52, 5);
insert into worldcitycoords values ('Bejing', 'China', 40, 116);
insert into worldcitycoords values ('Bombay', 'India', 19, 73);
insert into worldcitycoords values ('Calcutta', 'India', 22, 88);
insert into worldcitycoords values ('Caracas', 'Venezuela', 10, 67);
insert into worldcitycoords values ('Hong Kong', 'China', 22, 114);
insert into worldcitycoords values ('Lagos', 'Nigeria', 6, 3);
insert into worldcitycoords values ('Madrid', 'Spain', 40, 4 );
insert into worldcitycoords values ('Shanghai', 'China', 31, 121);
insert into worldcitycoords values ('Zurich', 'Switzerland', 47, 8);

select * from worldcitycoords;
```

Content

Output A9.14 *Content of Table WorldCityCoords*

CITY	COUNTRY	LATITUDE	LONGITUDE
Algiers	Algeria	37	3
Amsterdam	Netherlands	52	5
Bejing	China	40	116
Bombay	India	19	73
Calcutta	India	22	88
Caracas	Venezuela	10	67
Hong Kong	China	22	114
Lagos	Nigeria	6	3
Madrid	Spain	40	4
Shanghai	China	31	121
Zurich	Switzerland	47	8
Algiers	Algeria	37	3
Amsterdam	Netherlands	52	5
Bejing	China	40	116
Bombay	India	19	73
Calcutta	India	22	88
Caracas	Venezuela	10	67
Hong Kong	China	22	114
Lagos	Nigeria	6	3
Madrid	Spain	40	4
Shanghai	China	31	121
Zurich	Switzerland	47	8

WorldTemps

Code

```
create table worldtemps(City char(20),
      Country char(20),
      AvgHigh double,
      AvgLow double);

insert into worldtemps values ('Algiers', 'Algeria', 90, 45);
insert into worldtemps values ('Amsterdam', 'Netherlands', 79, 33);
insert into worldtemps values ('Bejing', 'China', 86, 17);
insert into worldtemps values ('Bombay', 'India', 90, 68);
insert into worldtemps values ('Calcutta', 'India', 97, 56);
insert into worldtemps values ('Caracas', 'Venezuela', 83, 57);
insert into worldtemps values ('Geneva', 'Switzerland', 76, 28);
insert into worldtemps values ('Hong Kong', 'China', 89, 51);
insert into worldtemps values ('Lagos', 'Nigeria', 90, 75);
insert into worldtemps values ('Madrid', 'Spain', 89, 36);
insert into worldtemps values ('Shanghai', 'China',., 33);
insert into worldtemps values ('Zurich', 'Switzerland', 78, 25);

select * from worldtemps;
```

Content

Output A9.15 *Content of Table WorldTemps*

CITY	COUNTRY	AVGHIGH	AVGLOW
Algiers	Algeria	90	45
Amsterdam	Netherlands	79	33
Bejing	China	86	17
Bombay	India	90	68
Calcutta	India	97	56
Caracas	Venezuela	83	57
Geneva	Switzerland	76	28
Hong Kong	China	89	51
Lagos	Nigeria	90	75
Madrid	Spain	89	36
Shanghai	China	.	33
Zurich	Switzerland	78	25

Appendix 5

DICTIONARY Table Descriptions

DICTIONARY.CATALOGS

The following table lists the columns that appear in the result table for the DICTIONARY.CATALOGS table.

Column Name	Column Number	Data Type	Description
CATALOG	1	NVARCHAR	Catalog name
DRIVER	2	NVARCHAR	Data source name
DESCRIPTION	3	NVARCHAR	Description of catalog

DICTIONARY.COLUMNS

The following table lists the columns that appear in the result table for the DICTIONARY.COLUMNS table.

Note: Null indicates either a null or a missing value.

Column Name	Column Number	Data Type	Description
TABLE_CAT	1	NVARCHAR[1]	Catalog name
TABLE_SCHEM	2	NVARCHAR	Schema name Null is returned if the schema is not applicable to the data source. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have schemas.
TABLE_NAME	3	NVARCHAR[1]	Table name
COLUMN_NAME	4	NVARCHAR[1]	Column name The driver returns an empty string for a column that does not have a name.
DATA_TYPE	5	signed INTEGER[1]	TKTS data type. This can be a SAS FedSQL data type or a remote table driver-specific data type. For datetime and interval data types, this column returns the concise data type such as DATE or INTERVAL_YEAR_TO_MONTH, rather than the nonconcise data type such as DATETIME or INTERVAL. For information about driver-specific TKTS data types, see the remote table driver's documentation.
TYPE_NAME	6	NVARCHAR[1]	Data source-dependent common data type name (for example, CHAR, NVARCHAR, or BIGINT).
COLUMN_SIZE	7	signed BIGINT[1]	If DATA_TYPE is CHAR or VARCHAR, this column contains the maximum length of the column in characters. For datetime data types, this is the total number of characters required to display the value when it is converted to characters. For numeric data types, this is either the total number of digits or the total number of bits allowed in the column, according to the NUM_PREC_RADIX column. For interval data types, this is the number of characters in the character representation of the interval literal.
BUFFER_LENGTH	8	signed BIGINT[1]	The length in bytes of data transferred. For numeric data, this size might be different from the size of the data stored on the data source.

Column Name	Column Number	Data Type	Description
			This value might be different from COLUMN_SIZE column for character data.
DECIMAL_DIGITS	9	signed INTEGER	The total number of significant digits to the right of the decimal point. If DATA_TYPE is TIME and TIMESTAMP, this column contains the number of digits in the fractional seconds component. For the other data types, this is the decimal digits of the column on the data source. For interval data types that contain a time component, this column contains the number of digits to the right of the decimal point (fractional seconds). For interval data types that do not contain a time component, this column is 0. Null is returned for data types where DECIMAL_DIGITS is not applicable.
NUM_PREC_RADIX	10	signed INTEGER	For numeric data types, either 10 or 2 is returned. If the value returned is 10, the values in COLUMN_SIZE and DECIMAL_DIGITS contain the number of decimal digits allowed for the column. For example, a DECIMAL(12,5) column would return 10 for NUM_PREC_RADIX, 12 for COLUMN_SIZE, and 5 for DECIMAL_DIGITS; a FLOAT column could return 10 for NUM_PREC_RADIX, 15 for COLUMN_SIZE, and null for DECIMAL_DIGITS. If the value returned is 2, the values in COLUMN_SIZE and DECIMAL_DIGITS give the number of bits allowed in the column. For example, a FLOAT column could return 2 for RADIX, 53 for COLUMN_SIZE, and null for DECIMAL_DIGITS. Null is returned for data types where NUM_PREC_RADIX is not applicable.
NULLABLE	11	signed INTEGER not null	TKTS_NO_NULLS if the column will not accept null values. TKTS _NULLABLE if the column accepts null values. TKTS_NULLABLE_UNKNOWN if it is not known whether the column accepts null values. The value returned for this column is different from the value returned for the IS_NULLABLE column. The NULLABLE column indicates with certainty that a column can accept null values, but cannot indicate with certainty that a column does not accept null values. The IS_NULLABLE

Column Name	Column Number	Data Type	Description
			column indicates with certainty that a column cannot accept null values, but cannot indicate with certainty that a column accepts null values.
REMARKS	12	NVARCHAR	A description of the column. For a FedSQL view, the value in the REMARKS column is FedSQL.VIEW.
COLUMN_DEF	13	NVARCHAR	The default value of the column. The value in this column should be interpreted as a string if it is enclosed in quotation marks. If null was specified as the default value, then this column is the word NULL, not enclosed in quotation marks. If the default value cannot be represented without truncation, then this column contains TRUNCATED, not enclosed in quotation marks. If no default value is specified, then this column is null. The value of COLUMN_DEF can be used in generating a new column definition, except when it contains the value TRUNCATED.
TKTS_DATA_TYPE	14	signed INTEGER[1]	Data type, as it appears in the TKTS_DESC_TYPE record field. This can be a SAS FedSQL data type or a remote table driver-specific data type. This column is the same as the DATA_TYPE column, with the exception of datetime and interval data types. This column returns the nonconcise data type (such as DATETIME or INTERVAL), rather than the concise data type (such as DATE or YEAR_TO_MONTH) for datetime and interval data types. If this column returns DATETIME or INTERVAL, the specific data type can be determined from the TKTS_DATETIME_SUB column. For information about driver-specific TKTS data types, see the remote table driver's documentation.
TKTS_DATETIME_SUB	15	signed INTEGER	The subtype code for datetime and interval data types. For other data types, this column returns a null.
CHAR_OCTET_LENGTH	16	signed BIGINT	The maximum length in bytes of a character or binary data type column. For all other data types, this column returns a null.

Column Name	Column Number	Data Type	Description
ORDINAL_POSITION	17	signed INTEGER[1]	The ordinal position of the column in the table. The first column in the table is number 1.
IS_NULLABLE	18	NVARCHAR[1]	NO if the column does not include nulls. YES if the column could include nulls. This column returns a zero-length string if nullability is unknown. ISO rules are followed to determine nullability. An ISO SQL-compliant DBMS cannot return an empty string. The value returned for this column is different from the value returned for the NULLABLE column.

1 Value cannot be a null

DICTIONARY.COLUMN_STATISTICS

The following table lists the columns that appear in the result table for the DICTIONARY.COLUMN_STATISTICS table.

Note: Null indicates either a null or a missing value.

Note: The first four columns do not appear in the result table.

Column Name	Column Number	Data Type	Description
TABLE_CAT	1	NVARCHAR[1]	Catalog name
TABLE_SCHEM	2	NVARCHAR	Schema name Null is returned if the schema is not applicable to the data source. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have schemas.
TABLE_NAME	3	NVARCHAR[1]	Table name

Column Name	Column Number	Data Type	Description
COLUMN_NAME	4	NVARCHAR[1]	Column name The driver returns an empty string for a column that does not have a name.
TYPE	5	unsigned INTEGER[1]	Type of information being returned: TKTS_COL_STAT indicates a statistic for the columns specified in the CARDINALITY and HISTOGRAM columns.
CARDINALITY	6	unsigned INTEGER	Cardinality of column or column set in table. This is the number of distinct, non-null values in the column or column set. Null is returned if the value is not available from the data source.
NULL_FRAC	7	DOUBLE	Fraction (expressed as a decimal) of the column's entries that are null. If any of the columns in a set of columns is null, then that entry is considered null.
MOST_COMMON_VALS	8	NVARCHAR	Contains a value that is common for this column. Null if there are no common values. Null if the number of columns is greater than one.
MOST_COMMON_FREQS	9	DOUBLE	A frequency of the common value returned in MOST_COMMON_VALS. This would likely be the number of occurrences of that particular column value divided by the total number of rows in the table. Null if the number of columns is greater than one.
HISTOGRAM_ENTRY	10	unsigned INTEGER	The entry for the HISTOGRAM_BOUNDS value. Null if the number of columns is greater than one.
HISTOGRAM_BOUNDS	11	NVARCHAR	One entry in a list of values that divide the column's values into groups of approximately equal population. The most common values (returned by MOST_COMMON_VALS) are omitted from the histogram calculation. No HISTOGRAM_BOUNDS are returned if the column's data type does not have a less than (<) operator, or if the values returned by

Column Name	Column Number	Data Type	Description
			MOST_COMMON_VALS account for the entire population of the column. Null if the number of columns is greater than one.
CORRELATION	12	DOUBLE	Statistical correlation between the physical row ordering and the logical ordering of the column values. The values range from -1 to +1. This value is not returned if the column's data type does not have a less than (<) operator. Null if the number of columns is greater than one.

1 Value cannot be a null or missing value

DICTIONARY.STATISTICS

The following table lists the columns that appear in the result table for the DICTIONARY.STATISTICS table.

Note: Null indicates either a null or a missing value.

Column Name	Column Number	Data Type	Description
TABLE_CAT	1	NVARCHAR[1]	TKTS catalog name.
TABLE_SCHEM	2	NVARCHAR	Schema name of the table to which the statistic or index applies. Null if not applicable to the data source. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have schemas.
TABLE_NAME	3	NVARCHAR not null	Name of the table to which the statistic or index applies.
NON_UNIQUE	4	signed INTEGER	Indicates whether the index prohibits duplicate values:

Column Name	Column Number	Data Type	Description
			TKTS_TRUE if the index values can be non-unique. TKTS_FALSE if the index values must be unique. Null is returned if TYPE is TKTS_TABLE_STAT.
INDEX_QUALIFIER	5	NVARCHAR	The identifier that is used to qualify the index name doing a DROP INDEX. Null is returned if an index qualifier is not supported by the data source or if TYPE is TKTS_TABLE_STAT. If a non-null value is returned in this column, it must be used to qualify the index name on a DROP INDEX statement; otherwise, the TABLE_SCHEM should be used to qualify the index name.
INDEX_NAME	6	NVARCHAR	Index name. Null is returned if TYPE is TKTS_TABLE_STAT.
TYPE	7	signed INTEGER[1]	Type of information being returned: TKTS_TABLE_STAT indicates a statistic for the table in the CARDINALITY or PAGES column. TKTS_INDEX_BTREE indicates a B-Tree index. TKTS_INDEX_CLUSTERED indicates a clustered index. TKTS_INDEX_CONTENT indicates a content index. TKTS_INDEX_HASHED indicates a hashed index. TKTS_INDEX_OTHER indicates another type of index.
ORDINAL_POSITION	8	signed INTEGER	Column sequence number in index starting with 1. Null is returned if TYPE is TKTS_TABLE_STAT.
COLUMN_NAME	9	NVARCHAR	Column name. If the column is based on an expression, such as SALARY + BENEFITS, the expression is returned; if the expression cannot be determined, an empty string is returned. Null is returned if TYPE is TKTS_TABLE_STAT.
ASC_OR_DESC	10	CHAR(1)	Sort sequence for the column: A for ascending; D for descending.

Column Name	Column Number	Data Type	Description
			Null is returned if column sort sequence is not supported by the data source or if TYPE is TKTS_TABLE_STAT.
CARDINALITY	11	signed BIGINT	Cardinality of table or index. Number of rows in table if TYPE is TKTS_TABLE_STAT. Number of unique values in the index if TYPE is not TKTS_TABLE_STAT. Null if the value is not available from the data source.
PAGES	12	signed INTEGER	Number of pages used to store the index or table. Number of pages for the table if TYPE is TKTS_TABLE_STAT. Number of pages for the index if TYPE is not TKTS_TABLE_STAT. Null if the value is not available from the data source or if not applicable to the data source.
FILTER_CONDITION	13	NVARCHAR	If the index is a filtered index, this is the filter condition, such as SALARY > 30000. If the filter condition cannot be determined, the value is an empty string. Null if the index is not a filtered index, if it cannot be determined whether the index is a filtered index, or if TYPE is TKTS_TABLE_STAT.
AVG_FANOUT[2]	14	DOUBLE	The average fan-out of internal nodes for an index. Valid for TKTS_INDEX_* TYPE records. Null if not known or available.
MAX_FANOUT[2]	15	unsigned INTEGER	The maximum fan-out of internal nodes for an index. Valid for TKTS_INDEX_* TYPE records. Null if not known or available.
MIN_FANOUT[2]	16	unsigned INTEGER	The minimum fan-out of internal nodes for an index. Valid for TKTS_INDEX_* TYPE records. Null if not known or available.
INDEX_LEVELS[2]	17	unsigned INTEGER	The number of levels in an index. Valid for TKTS_INDEX_* TYPE records. Null if not known or available.

Column Name	Column Number	Data Type	Description
LEAF_LEVEL_BLOCKS[2]	18	unsigned INTEGER	The number of blocks at the leaf level of an index. Valid for TKTS_INDEX_* TYPE records. Null if not known or available.
LOCAL[2]	19	NVARCHAR	Whether a table is local or must be accessed over a network. Valid for TKTS_TABLE_STAT TYPE records. Values YES, NO, or Null if unknown.
PARTITION_SCHEME[2]	20	unsigned INTEGER	Whether the table is partitioned and, if so, does it use a round-robin, hash- or range-partitioning scheme. Valid for TKTS_TABLE_STAT TYPE records. TKTS_PARTITION_NONE indicates the table is not partitioned. TKTS_PARTITION_ROUND_ROBIN indicates a round-robin partitioning scheme. TKTS_PARTITION_HASH indicates a hash partitioning scheme. TKTS_PARTITION_RANGE indicates a range partitioning scheme. Null if not known or available.
FRAGMENTATION[2]	21	unsigned INTEGER	If the data is distributed, indicates whether it uses vertical fragmentation, horizontal fragmentation, derived fragmentation, or a hybrid fragmentation. Valid for TKTS_TABLE_STAT TYPE records. TKTS_FRAG_NONE indicates the data is not distributed. TKTS_FRAG_VERTICAL indicates vertical fragmentation. TKTS_FRAG_HORIZONTAL indicates horizontal fragmentation. TKTS_FRAG_DERIVED indicates derived fragmentation. TKTS_FRAG_HYBRID uses hybrid fragmentation. Null if not known or available.
LABEL[2]	22	NVARCHAR (max 256 characters)	Label associated with the table. Valid for TKTS_TABLE_STAT TYPE records. Null if not known or available.
APPLICATION_TYPE[2]	23	NVARCHAR (max 8 characters)	SAS application-specific type associated with the table. The value of this type field is created, stored, and retrieved by some SAS applications. The meaning of the value is specific to the application that created it. Valid for TKTS_TABLE_STAT TYPE records.

Column Name	Column Number	Data Type	Description
			Null if not known or available.
PROTECTED[2]	24	signed INTEGER	Indicates whether table is password protected. Valid for TKTS_TABLE_STAT TYPE records. TKTS_PROTECTION_NONE indicates table is not password protected. TKTS_PROTECTION_READ indicates table is Read protected. TKTS_PROTECTION_WRITE indicates table is Write protected. TKTS_PROTECTION_ALTER indicates table is Alter protected. TKTS_PROTECTION_ENCRYPTED indicates table is encrypted. Null if not known or available.
COMPRESS[2]	25	NVARCHAR (max 8 characters)	Indicates whether the records in the table are compressed. Valid for TKTS_TABLE_STAT TYPE records. NO Indicates records are uncompressed. YES|CHAR indicates RLE(Run Length Encoding) used to compress records. BINARY indicates RDC(Ross Data Compression) used to compress records. Null if not known or available.
CHAR_CEI[2]	26	unsigned INTEGER	Encoding value used for the table. Valid for TKTS_TABLE_STAT TYPE records. Null if not known or available.
DELETED_ROWS[2]	27	signed BIGINT	Number of deleted rows in the table. Valid for TKTS_TABLE_STAT TYPE records. Null if not known or available.
DATE_CREATED[2]	28	DOUBLE	Date the table was created. Value is a DATETIME that is defined by SAS. Valid for TKTS_TABLE_STAT TYPE records. Null if not known or available.
DATE_MODIFIED[2]	29	DOUBLE	Date the table was last modified. Value is a DATETIME that is defined by SAS. Valid for TKTS_TABLE_STAT TYPE records. Null if not known or available.

Column Name	Column Number	Data Type	Description
TABLE_ATTRIBUTES[2]	30	signed INTEGER	Attributes that apply to the table. This column represents a set of flags to convey specific attributes. A value of 0 for any flag indicates either a False or unknown condition. Valid for TKTS_TABLE_STAT TYPE records. Null if not known or available.
ROW_LENGTH[2]	31	signed BIGINT	Length of the rows in the table on disk. Valid for TKTS_TABLE_STAT TYPE records. Null if not known or available.

1 Value cannot be a null or missing value
2 Optional-not defined in ODBC

DICTIONARY.TABLES

The following table lists the columns that appear in the result table for the DICTIONARY.TABLES table.

Note: Null indicates either a null or a missing value.

Column name	Column number	Data type	Description
TABLE_CAT	1	NVARCHAR not Null	Catalog name
TABLE_SCHEM	2	NVARCHAR	Schema name; null if not applicable to the data source. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have schemas.
TABLE_NAME	3	NVARCHAR not null	Table name
TABLE_TYPE	4	NVARCHAR not null	Table type name; one of the following: TABLE, VIEW, SYSTEM TABLE, GLOBAL TEMPORARY, LOCAL TEMPORARY, ALIAS, SYNONYM, or a data source-specific type name. The meanings of ALIAS and SYNONYM are driver-specific.

Column name	Column number	Data type	Description
REMARKS	5	NVARCHAR	A description of the table
NATIVE_CAT	6	NVARCHAR	The native data store's Catalog; null if not applicable to the data source. If a driver supports catalogs for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have catalogs.
SECURITY	7	NVARCHAR	The security environment the view executes under. Valid values are “DEFINER”, “INVOKER”, or null. A value of “DEFINER” indicates the view executes under the rights of the owner of the schema containing the view. A value of “INVOKER” indicates the view executes under the rights of the invoker. Null is returned if the security environment is unknown or the view is a native view.
SERVICE	8	NVARCHAR	The service responsible for managing the view. Valid values are “FEDSQL”, “DRIVER”, or null. A value of “FEDSQL” indicates the view is managed via a FEDSQL connection to the appropriate driver. A value of “DRIVER” (or a null value) indicates a native view managed by a DBMS driver.

Appendix 6

How to Perform Common PROC SQL Tasks with FedSQL

About Performing PROC SQL Tasks with FedSQL

This appendix describes how popular PROC SQL tasks that rely on non-ANSI functionality can be performed with FedSQL. It also describes quoting differences between the two languages. To create the tables used in the examples, see "Code to Create the Sample Tables" on page 1220. The examples assume that the source tables are stored in a location that has been assigned by using the libref SQL.

Overwrite an Existing Table with FedSQL

When creating tables with FedSQL, you cannot overwrite an existing table by default. You must destroy the table with the DROP TABLE statement first. To drop an existing table in the same procedure request as the CREATE TABLE statement, specify the FORCE option in the DROP TABLE statement as follows:

```
proc fedsql;
   drop table test force;
   create table test(col1 char(5), col2 int);
   insert into test  values ('high', 5);
quit;
```

The FORCE option suppresses completion messages that, when FORCE is omitted, cause the CREATE TABLE statement to fail.

In CAS, you can overwrite an existing session table by specifying the REPLACE= table option in the CREATE TABLE statement. Here is an example:

```
cas mysess;
proc fedsql sessref=mysess;
  create table test2 {options replace=true} as
  select * from test1;
quit;
```

Defining SAS Formats, Informats, and Labels When Creating a Table

Note: The information in this topic applies to FedSQL in SAS 9.4. By contrast, FedSQL in CAS supports creation of FedSQL output tables from existing input tables only.

FedSQL supports SAS FORMAT, INFORMAT, and LABEL modifiers for data definitions in the CREATE TABLE statement. Here is an example of how the modifiers are specified in FedSQL.

Note: When executing the code in this example and the first example in "Inserting Values into a Table" on page 1204, be sure to use the Work library. Otherwise, you can mistakenly overwrite the Countries table that you downloaded.

```
proc fedsql;
   create table countries
     (
      Name char(35) having format $35. informat $35. label 'Name',
     Capital char(35) having format $35. informat $35. label 'Capital',
     Population double having format comma15. informat comma15. label
'Population',
     Area double having format comma10. informat comma10. label 'Area',
     Continent char(30) having format $30. informat $30. label
'Continent',
     UNDate double having format year4. label 'UNDate'
  );
quit;
```

For comparison, here is an example of how the modifiers are specified in PROC SQL.

```
proc sql;
   create table countriesA
     (
      Name char(35) format=$35. informat=$35. label="Name",
      Capital char(35) format=$35. informat=$35. label="Capital",
      Population num format=comma15. informat=comma15.
label="Population",
      Area num format=comma10. informat=comma10. label="Area",
      Continent char(30) format=$30. informat=$30. label="Continent",
      UNDate num format=year4. label="UNDate"
  );
quit;
```

Note the following points about the modifiers in FedSQL:

- The FORMAT, INFORMAT, and LABEL modifiers are specified in a HAVING clause.

- Values are separated from their modifiers with a blank value instead of an equal sign.
- The label value must be enclosed in single quotation marks.

When you specify a format or informat, if the data type is not either CHAR nor DOUBLE, it is converted to either CHAR or DOUBLE.

FedSQL applies formats when data is selected for display for data sources where format catalogs are available. The informat is not applied to the data; it is for information only. The client application applies the informat to the data.

Inserting Values into a Table

Both SAS SQL and FedSQL support a VALUES clause and query expressions in the INSERT statement. However, a FedSQL INSERT statement can contain a single VALUES clause. FedSQL does not supported stacked VALUES clauses. A request similar to the following returns a syntax error in FedSQL:

```
proc sql;
   insert into countries
           values ('Afghanistan', 'Kabul', 17070323, 251825, 'Asia',
1946)
           values ('Albania', 'Tirane', 3407400, 11100, 'Europe', 1955)
           values ('Algeria', 'Algiers', 28171132, 919595, 'Africa',
1962)
           values ('Andorra', 'Andorra la Vella', 64634, 200,
'Europe', 1993);
quit;
```

To accomplish the same result in FedSQL, you must submit multiple INSERT statements:

```
proc fedsql;
   insert into countries values ('Afghanistan', 'Kabul', 17070323,
251825,
    'Asia', 1946);
   insert into countries values ('Albania', 'Tirane', 3407400, 11100,
    'Europe', 1955);
   insert into countries values ('Algeria', 'Algiers', 28171132,
919595,
    'Africa', 1962);
   insert into countries values ('Andorra', 'Andorra la Vella', 64634,
200,
    'Europe', 1993);
quit;
```

You can also insert table values by using a query expression with FedSQL. Consider the following example:

```
proc fedsql;
   create table sql.newcountries
     (
      Name char(35) having format $35. informat $35. label 'Name',
     Capital char(35) having format $35. informat $35. label 'Capital',
```

```
     Population double having format comma15. informat comma15. label
'Population',
     Area double having format comma10. informat comma10. label 'Area',
     Continent char(30) having format $30. informat $30. label
'Continent',
     UNDate double having format year4. label 'UNDate'
  );
  title "World's Largest Countries";
  insert into newcountries
     select * from sql.countries
     where population >= 130000000;
   select * from sql.newcountries;
quit;
```

The example defines a new table named Newcountries and submits a query expression in the INSERT statement to populate the table.

World's Largest Countries

Name	Capital	Population	Area	Continent	UNDate
Brazil	Brasilia	160,310,357	3,286,500	South America	1945
China	Beijing	1,202,215,077	3,696,100	Asia	1945
India	New Delhi	929,009,120	1,222,600	Asia	1945
Indonesia	Jakarta	202,393,859	741,100	Asia	1950
Russia	Moscow	151,089,979	6,592,800	Europe	1945
United States	Washington	263,294,808	3,787,318	North America	1945

Table Newcountries could also have been created by using the FedSQL CREATE TABLE statement with the AS expression. Here is an example:

```
proc fedsql;
title "World's Largest Countries";
create table newcountries2 as
select * from sql.countries
where population >= 130000000;
select * from newcountries;
quit;
```

However, FedSQL does not support the use of FORMAT and LABEL modifiers in a query expression. If it is important to define formats and labels for a table, define the table with the modifiers first and insert the data with a query expression as a separate step. FedSQL preserves formats and labels from an existing table when reading the table.

There is an important difference from PROC SQL to note about FedSQL query expressions: PROC SQL supports symbols and mnemonics for comparison operators. FedSQL supports operators only. For a listing of supported operators, see “<sql-expression> Expression” on page 894.

Specifying SAS Formats and Labels in a Query Expression

Note: The information in this topic applies to FedSQL use in SAS 9.4 and in CAS.

To format a column in the SELECT statement, apply a temporary format by using the PUT function. Formats that are applied with the PUT function are supported for all FedSQL data sources. For a listing of the formats supported in the PUT function, see "Formats Supported with the PUT Function, by Category" on page 85.

To label a column in the SELECT statement, specify a column alias in the place of a label.

The following example illustrates how formats and aliases can be specified in a query expression:

```
proc fedsql;
title 'Bonus Information';
 select IdNumber as "IdNumber",
        put(Salary,dollar8.) as "Salary",
        put(Salary*.025, dollar8.) as "Bonus"
from sql.payroll limit 10;
quit;
```

When you apply a function on a column and an alias is not specified for the column, the stored column name is the literal "column" (for the PUT function) or the name of the function. A column alias enables you to specify a more meaningful name.

> **TIP** If you use a quoted alias for a computed column, consider defining quoted aliases for other columns as well. FedSQL stores regular identifiers as uppercase. It stores delimited identifiers as they are specified. A column alias is displayed as uppercase unless you quote the alias. Use of column aliases ensures that the letter casing of column names in output is consistent.

Note the following:

- In FedSQL, a column alias must be preceded by the AS keyword.
- When quotation marks are used with a column alias, they must be double quotation marks.

Rewriting a GROUP BY Remerge Query in FedSQL

Note: The information in this topic applies to FedSQL use in SAS 9.4 and in CAS.

FedSQL does not support PROC SQL remerge functionality. Remerge is not an ANSI concept. Remerge occurs whenever any of the following conditions exist in a PROC SQL request:

- The SELECT clause references a column name that is not in an aggregate function or other column names that are not specified in the GROUP BY clause.
- The ORDER BY clause references a column name that is not referenced by the SELECT clause.

Another way to recognize a remerge query is that PROC SQL writes the following message to the log: `NOTE: The query requires remerging summary statistics back with the original data.`

Remerge queries perform a statistical analysis and integrate the results of the analysis with other columns from the data.

In a GROUP BY query, remerge is a useful feature when you want to return only the extreme rows from data based on a set of groupings. For example, you might have rows describing only the closest (or farthest) item in one or more areas, the oldest (or youngest) person in one or more groupings, or the most popular (or least popular) items in one or more groupings.

Here is an example of a PROC SQL query that uses remerge. The query identifies the oldest employee of each gender in a payroll table. The extreme condition in the query is earliest birthdate. The grouping criterion is gender.

```
proc sql;
   title 'Oldest Employee of Each Gender';
   select *
      from sql.payroll
      group by gender
      having birth=min(birth);
quit;
```

Note that this query references column names (by means of the SELECT * statement) that are neither in an aggregate function nor included in the GROUP BY statement.

Figure A11.1 Result of PROC SQL Remerge Query on Table Payroll

Oldest Employee of Each Gender

IdNumber	Gender	Jobcode	Salary	Birth	Hired
1935	F	NA2	51081	28MAR54	16OCT81
1118	M	PT3	11379	16JAN44	18DEC80

The output table includes values for all columns in table Payroll for the oldest female and oldest male in the Payroll table. The oldest female was born on March 28, 1954. The oldest male was born on January 16, 1944.

To write a GROUP BY remerge request using FedSQL:

1. Identify the extreme value and the groupings for your query.
2. Select the rest of the row based on the results of the subquery, reversing the requirements that remerge was meant to address for the GROUP BY clause.

To rewrite the example above in FedSQL:

1. Identify the earliest birthdates in table Payroll by gender:

```
proc fedsql;
title 'Earliest Birthdate by Gender';
  select Gender, min(Birth) from sql.payroll
  group by Gender;
quit;
```

This is your subquery.

Figure A11.2 Result of FedSQL Query to Identify Earliest Birthdate by Gender

Earliest Birthdate By Gender

Gender	MIN
F	28MAR1954
M	16JAN1944

2. Specify the subquery and retrieve other column values from the original data for the rows returned by the subquery.

```
proc fedsql;
   title 'Oldest Employee of Each Gender';
  select 2
    p.IdNumber, p.Gender, p.Jobcode, p.Salary, p.Birth, p.Hired
  from
    (
     select Gender, min(Birth) as min_birth 1
     from sql.payroll
     group by Gender
     ) as t
```

```
   inner join sql.payroll as p on p.Gender=t.gender and
p.Birth=t.min_birth; 3
quit;
```

1. The subquery is executed first. It performs the statistical analysis and creates the groupings. Column Birth is the analysis column. Column Gender is the grouping column.
2. The outer query explicitly specifies the names of the columns to include in the output.
3. The grouping column, Gender, is included in the inner query, the outer query, and again as a join condition in addition to being specified as the grouping column. The analysis column, Birth, is specified in the outer query and also as a join condition.

Here are the results of the query. The FedSQL query returns the same results as the PROC SQL remerge query.

Figure A11.3 *Result of FedSQL Subquery and Self-join*

Oldest Employee of Each Gender

IdNumber	Gender	Jobcode	Salary	Birth	Hired
1935	F	NA2	51081	28MAR54	16OCT81
1118	M	PT3	11379	16JAN44	18DEC80

Here is an example of a remerge query that involves two tables. The Stores table contains information about the stores in a given area. The Houses table contains information about the houses in the same area. The PROC SQL query uses a calculated column from the SELECT statement as a grouping condition in the HAVING clause to identify the store that is closest to each house.

```
proc sql;
   title 'Each House and the Closest Store';
   select house, store, sqrt((abs(s.x-h.x)**2)+(abs(h.y-s.y)**2)) as dist
      from sql.stores s, sql.houses h
group by house
      having dist=min(dist);
quit;
```

Figure A11.4 *Result of PROC SQL Remerge Query on the Sales and Houses Tables*

Each House and the Closest Store

House	Store	Dist
house1	store1	4
house2	store2	2
house2	store3	2
house3	store3	2.236068
house4	store4	2

To write the query in FedSQL:

1 Create the subquery. In this query, the extreme condition is nearest store distance. The groupings are by house. The following code calculates the distance between the houses and the stores, defines the Min_Dist column to store the results, and groups the results by house.

```
proc fedsql;
title 'House and Minimum Distance';
   select house, min(sqrt((abs(s.x-h.x)**2)+(abs(h.y-s.y)**2))) as min_dist
     from sql.stores s, sql.houses h
   group by house;
quit;
```

Figure A11.5 *Result of FedSQL Subquery on the Houses and Stores Tables*

House and Minimum Distance

House	MIN_DIST
house1	4
house2	2
house3	2.236068
house4	2

2 Execute the subquery and retrieve other column values from the two tables for the rows returned by the subquery.

```
proc fedsql;
title 'Each House and the Closest Store';
select 2
  h.house, s.store, t.min_dist as "Dist"
from
  sql.houses h
cross join
  sql.stores s
inner join
  /* minimum distance from each house to any store */
```

```
  ( 1
    select house, min(sqrt((abs(s.x-h.x)**2)+(abs(h.y-s.y)**2))) as
min_dist
    from stores s, houses h
    group by house 3
  ) as t
on h.house = t.house
where
  sqrt((abs(s.x-h.x)**2)+(abs(h.y-s.y)**2)) = t.min_dist;
quit;
```

1 Again, the subquery is executed first. It performs the statistical analysis and creates the groupings. Calculated column Min_Dist is the analysis column. House is the grouping column.

2 The outer query specifies to display the House column from the Houses table, the Store column from the Stores table, and the Min_Dist column from the subquery.

3 The grouping column House is specified in the inner query, the outer query, and as the join condition in addition to the GROUP BY statement. The analysis column Min_Dist is specified in the outer query and in the WHERE clause. The WHERE clause specifies to retrieve only values for rows that meet the requirements of the distance calculation (the rows that identify the nearest store to each house).

Figure A11.6 *Result of the FedSQL Subquery and Joins on the Stores and Houses Tables*

Each House and the Closest Store

House	Store	Dist
house1	store1	4
house2	store2	2
house2	store3	2
house3	store3	2.236068
house4	store4	2

Rewriting a Query That Specifies the CALCULATED Keyword

Note: The information in this topic applies to FedSQL use in SAS 9.4 and in CAS.

FedSQL does not support the CALCULATED keyword in query expressions. The CALCULATED keyword is not an ANSI concept. The CALCULATED keyword is similar to an in-line view. However, instead of defining a temporary alias for a column, the keyword defines a shortcut to a previous SQL expression. You can use it to reference column aliases that are associated with calculated expressions. Consider the following example that specifies the CALCULATED keyword.

```
proc sql;
title 'Total First Quarter Sales';
   select sum(January)  as JanTotal,
          sum(February) as FebTotal,
          sum(March)    as MarTotal,
          sum(calculated JanTotal, calculated FebTotal,
calculated MarTotal) as GrandTotal format=dollar10.
from sql.Sales;
quit;
```

In the preceding query, the CALCULATED keyword is used to reference the previous statements as follows:

- the first instance of the CALCULATED keyword references the expression select sum(January)
- the second instance references the expression select sum(February)
- the third instance references the expression select sum(March)
- to enable the computation sum(January) + sum(February) + sum(March) = GrandTotal.

Figure A11.7 *Result of SQL Query with the CALCULATED Keyword*

Total First Quarter Sales

JanTotal	FebTotal	MarTotal	GrandTotal
6800	7500	8100	$22,400

To write the preceding query using FedSQL, repeat the calculated expression:

```
proc fedsql;
title 'Total First Quarter Sales';
   select sum(January)  as JanTotal,
          sum(February) as FebTotal,
          sum(March)    as MarTotal,
          put(sum(January) + sum(February) + sum(March), dollar10.) as GrandTotal
from sql.Sales;
quit;
```

Total First Quarter Sales

JANTOTAL	FEBTOTAL	MARTOTAL	GRANDTOTAL
6800	7500	8100	$22,400

A better way to write the FedSQL request, however, is as a subquery:

```
   proc fedsql;
```

```
select
  t.JanTotal,
  t.FebTotal,
  t.MarTotal,
  put(sum(t.JanTotal, t.FebTotal, t.MarTotal), dollar10.) as
"GrandTotal"
from
  (select sum(January) as "JanTotal",
          sum(February) as "FebTotal",
          sum(March) as "MarTotal"
   from sql.Sales) as t;
quit;
```

When you write the request as a subquery, you can reference each expression by using its column alias instead of repeating the expression. Use of a subquery is more efficient, and it is also safer to use when the SQL expressions are complex.

Specifying Multiple Arguments in an SQL Expression

Note: The information in this topic applies to the use of FedSQL in a SAS library and in CAS.

When more than one condition is specified in the WHEN argument of a CASE expression in PROC SQL, the conditions are linked with an implicit AND operator. Consider the following PROC SQL example, which queries the table WorldCityCoords to label the climate zones of world cities with a descriptor.

```
proc sql outobs=12;
   title 'Climate Zones of World Cities';
   select City, Country, Latitude,
          case
             when Latitude gt 67 then 'North Frigid'
             when 67 ge Latitude ge 23 then 'North Temperate'
             when 23 gt Latitude gt -23 then 'Torrid'
             when -23 ge Latitude ge -67 then 'South Temperate'
             else 'South Frigid'
          end as ClimateZone
      from sql.worldcitycoords
      order by City;
quit;
```

Climate Zones of World Cities

City	Country	Latitude	ClimateZone
Abadan	Iran	30	North Temperate
Acapulco	Mexico	17	Torrid
Accra	Ghana	5	Torrid
Adana	Turkey	37	North Temperate
Addis Ababa	Ethiopia	9	Torrid
Adelaide	Australia	-35	South Temperate
Aden	Yemen	13	Torrid
Ahmenabad	India	22	Torrid
Algiers	Algeria	37	North Temperate
Alice Springs	Australia	-24	South Temperate
Amman	Jordan	32	North Temperate
Amsterdam	Netherlands	52	North Temperate

PROC SQL processes the highlighted sections of the code as shown below:

- When 67 is greater than or equal to Latitude (and) Latitude is greater than or equal to 23, specify 'North Temperate'.
- When 23 is greater than Latitude (and) Latitude is greater than -23, specify 'Torrid'.
- When -23 is greater than or equal to Latitude (and) Latitude is greater than or equal to -67, specify 'South Temperate'.
- Otherwise, specify “South Frigid”.

By contrast, when more than one condition is specified in the WHEN argument of a CASE expression in FedSQL, the highlighted sections of code are processed as shown below:

- The conditions must be equal.
- An operator must specify the comparison that is to be made between the conditions.

Here is an example:

```
proc fedsql;
   title 'Climate Zones of World Cities';
   select City, Country, Latitude,
          case
             when Latitude > 67 then 'North Frigid'
             when (67 >= Latitude) and (Latitude >= 23) then 'North Temperate'
             when (23 > Latitude) and (Latitude > -23) then 'Torrid'
             when (-23 >= Latitude) and (Latitude >= -67) then 'South Temperate'
             else 'South Frigid'
          end as "ClimateZone"
      from sql.worldcitycoords
      order by City limit 12;
quit;
```

Note that PROC SQL's mnemonics also must be converted to operators to run in FedSQL. FedSQL requires the use of operators.

Using Explicit Pass-Through

Both FedSQL and SAS SQL enable you to submit DBMS-specific SQL statements to the DBMS by using explicit SQL pass-through. Non-query DBMS SQL statements are submitted to the DBMS by using the EXECUTE statement. Queries can be submitted in the CONNECTION TO component of the SELECT statement FROM clause.

In SAS SQL, statements requesting explicit pass-through are used in conjunction with the CONNECT and DISCONNECT statements. Here is an example:

```
proc sql;
connect to oracle as ora2 (user=user-id password=password);
execute (create table new (a NUMBER, b TIMESTAMP, c VARCHAR2(15))) by
ora2;
execute(insert into new values(12345, date'2003-11-22', 'John Doe'))
by ora2;
select * from connection to ora2 (select a, b, c from new);
disconnect from ora2;
quit;
```

In FedSQL, there is no need to specify CONNECT and DISCONNECT statements. The data source connection is established with a libref (or a caslib in CAS). Here is an example:

```
libname ora2 oracle ...;
proc fedsql;
execute (create table new2 (a NUMBER, b TIMESTAMP, c VARCHAR2(15))) by
ora2;
execute(insert into new2 values(12345, date'2003-11-22', 'John Doe'))
by ora2;
select * from connection to ora2(select a, b, c from new2);
quit;
```

Note: Currently, only the SELECT from CONNECTION TO explicit pass-through syntax is supported for FedSQL in CAS.

However, in FedSQL, you can more easily create, populate, and query tables in a DBMS by using FedSQL statements than by submitting native statements using explicit pass-through. FedSQL is a vendor-neutral SQL dialect that accesses data from various data sources without having to submit queries in the SQL dialect that is specific to the data source. It provides ANSI compliant data types for writing DBMS data and, when reading DBMS data, it maps non-ANSI data types to ANSI types internally. For more information about SAS and third-party data type mappings, see Appendix 2, "Data Type Reference," on page 1109.

For example, the following FedSQL statements can create a table in any DBMS that is supported by FedSQL.

```
proc fedsql;
create table ora2.new3(a decimal(10,2), b timestamp, c varchar(15));
```

```
insert into ora2.new3 values (12345, timestamp(2003-11-22 02:25:00),
'John Doe');
select * from ora2.new3;
quit;
```

Data type conversions are processed internally by FedSQL. In addition, FedSQL supports full and partial implicit pass-through for third-party data sources by default.

Note: Partial FedSQL pass-through is not yet available for FedSQL in CAS.

Comparing Implicit SQL Pass-Through

Both PROC FEDSQL and SAS SQL are capable of passing requests to the DBMS for processing via implicit SQL pass-through.

SAS SQL supports implicit pass-through when the IPASSTHRU statement option is set. IPASSTHRU is set by default. PROC SQL statement options, SAS system options, and SAS/ACCESS LIBNAME statement options are available to control aspects of the pass-through process such as the following:

- how many times SAS SQL retries a query when an implicit SQL pass-through fails (SQLIPONEATTEMPT statement option and system option)
- how formats and observations are handled (SQLREDUCEPUTVALUES= and SQLREDUCEPUTOBS= SQL statement options and system options)
- which functions are passed down to the DBMS for processing (SQL_FUNCTIONS=)
- whether the SAS/ACCESS engine generates database-specific queries for CREATE TABLE, INSERT, and DELETE FROM TABLE statement requests, or whether SAS SQL statements are used (DBIDIRECTEXEC= system option)

FedSQL queries are automatically evaluated for implicit pass-through based on the following rules:

- FedSQL passes single source queries to the underlying data source unless the query includes one of the following:
  - functions that are specific to SAS such as PUT
  - certain aggregate statistics such as SKEWNESS, STUDENTS_T, NMISS, KURTOSIS, CSS, USS, and PROBT
  - mathematical functions such as SIN, COS, ATAN, and TAN
  - ANSI-compliant FedSQL syntax that might prevent IP if the data source is not ANSI compliant in that area

  When full-query single source pass-through is not possible, the predicates (for example, the WHERE clauses) are passed down to subset the rows that must be transported into FedSQL for local processing.
- When FedSQL processes queries that include multiple data sources, it accomplishes implicit pass-through by breaking the query into multiple queries and passing these individual queries to their respective drivers. The queries

must meet the requirements for single-source implicit pass-through. In addition, multiple source IP has the following additional limitations:

- A maximum of ten tables can be in one comma join.
- The tables in each side of a set operation (for example, UNION, INTERSECT, and EXCEPT) must share the same data source for multiple source IP to perform correctly. Here is an example:

  ```
  /* join of two Teradata tables */
  select t.col1, t.col2, t1.col1, t1.col4
  from tera.employees t,  tera.employees t1
  where t.col1=t1.col1
  Union
  /* join of two Oracle tables */
  select b.col1, b.col2, b1.col1, b1.col4
  from ora.test1 b, ora.test2 b1
  where b.col1=b1.col1;
  ```

  If the tables from either side of the set operation are from multiple data sources, then partial implicit pass-through might not occur.
- If the query contains a correlated subquery, no multiple source IP is attempted.
- Database tables must be in the same catalog in order to be merged and joined using implicit pass-through. Merges and joins of database tables that exist in different catalogs are processed on the client. A query that contains references to multiple schemas (but only one catalog) can be passed to the data source.

Note: Neither partial FedSQL implicit pass-through nor multi-source FedSQL implicit pass-through are currently supported in CAS.

FedSQL writes messages to the log that indicate when implicit SQL pass-through is used.

Using SAS Macros with FedSQL

Note: The information in this topic applies to FedSQL use in SAS 9.4.

Macros that are used by PROC SQL can be executed from PROC FEDSQL by modifying the macro definition to meet FedSQL semantics. Consider the following example from the PROC SQL documentation that has been modified for use in PROC FEDSQL. In the example, table Survey contains data from a questionnaire about diet and exercise habits. SAS enables you to use a special notation for missing values. In the EDUC column of table Survey, note the following:

- The .x notation indicates that the respondent gave an answer that is not valid.
- An .n indicates that the respondent did not answer the question.
- A period (.) that is used as a missing value indicates a data entry error.

The macro counts the number of rows that have each value and returns the information in a table.

Here is the PROC SQL code:

```
%macro countm(col);
    count(&col) "Valid Responses for &col",
nmiss(&col) "Missing or NOT VALID Responses for &col",
count(case
            when &col=.n  then "count me"
            end) "Coded as NO ANSWER for &col",
   count(case
            when &col=.x  then "count me"
            end) "Coded as NOT VALID answers for &col",
   count(case
            when &col=.  then "count me"
            end) "Data Entry Errors for &col"
%mend;

proc sql;
   title 'Counts for Each Type of Missing Response';
   select count(*)  "Total No. of Rows",
          %countm(educ)
      from sql.survey;
quit;
```

Here are the macro definition and the query, modified for FEDSQL:

```
%macro countm_fedsql(col);
  count(&col) as 1 "Valid Responses for &col",
  nmiss(&col) as "Missing or NOT VALID Responses for &col",
  count(case
            when &col=.n  then 'count me' 2
            end) as "Coded as NO ANSWER for &col",
  count(case
            when &col=.x  then 'count me'
            end) as "Coded as NOT VALID answers for &col",
  count(case
            when &col=.  then 'count me'
            end) as "Data Entry Errors for &col"
%mend;

proc fedsql;
   title 'Counts for Each Type of Missing Response';
   select count(*) as "Total No. of Rows", 3
          %countm_fedsql(educ)
      from sql.survey;
quit;
```

1 FedSQL requires the keyword AS between a column expression and its alias. To preserve case, an alias must be double-quoted.

2 FedSQL requires the character constant 'count me' to be single-quoted, not double-quoted.

3 FedSQL specifies a replacement column name as a column alias instead of applying a label.

Comparison of FedSQL and PROC SQL Quoting Requirements

Table A11.1 *FedSQL and PROC SQL Quoting Differences*

Task	FedSQL	PROC SQL
Preserve the case, spacing, or use of special characters in an identifier (delimited identifier)	Identifiers are stored as uppercase by default; use double quotation marks to preserve casing or to include spaces or special characters. **Note:** When a delimited identifier is used, FedSQL preserves casing; it does not enforce case sensitivity. The target data source determines whether values that differ only in case are seen as the same or different.	The casing of identifiers is preserved by default. Use of quotation marks returns an error. The identifiers must follow rules for SAS names, unless PROC SQL is communicating with a SAS/ACCESS engine that supports the PRESERVE_COL_NAMES= and PRESERVE_MEM_NAMES= options. When these options are set to YES, the data source enforces its own case sensitivity rules.

Preserve the case of a column alias (delimited alias)	Column aliases are displayed as uppercase by default. To preserve casing or use spaces or special characters, enclose the alias in double quotation marks.
	Examples:
	`as "ListPrice"`
	`as "Data Entry Errors for &col"`
		Column aliases are displayed as specified. If the alias includes a space or a special character other than an underscore (_), use double quotation marks.
		Examples:
		`as ListPrice`
		`as "Data Entry Errors for &col"`

Specify a value for the LABEL modifier	Label values must be specified within single quotation marks.
	Example:
	`label 'Continent'`
		Label values can be specified within single or double quotation marks.
		Examples:
		`label="Continent"`
		`label='Continent'`

Refer to a character constant	Character constants must be specified within single quotation marks.
	Example:
	`where col1='count me'`
		Character constants can be specified within single or double quotation marks.
		Example:
		`where col1='count me'`
		`where col1="count me"`

Escape a reserved word	Reserved words can be escaped by enclosing them within double quotation marks. Example: `"CASE"` See "FedSQL Reserved Words" on page 71.	If you have a column named CASE in a table and you want to specify it in a PROC SQL step, you can use the SAS data set option RENAME= to rename that column for the duration of the query. You can enclose CASE in double quotation marks ("CASE") and set the PROC SQL procedure option DQUOTE=ANSI.

The following example illustrates use of a delimiter with the reserved word CASE:

```
proc fedsql;
create table test ("Case" char(4));
insert into test values ('test');
title 'Reading a Reserved Word with FedSQL';
select "Case" from test;
quit;
```

Reading a Reserved Word with FedSQL

Case
test

Code to Create the Sample Tables

Countries and WorldCityCoords Tables

Because of the tables' size, the code to create the Countries table that is used in "Inserting Values into a Table" on page 1204 and the WorldCityCoords table used in "Specifying Multiple Arguments in an SQL Expression" on page 1213 is not provided here. These tables are available as a download from http://support.sas.com/

documentation/onlinedoc/base/index.html. Under the “Related Documentation” topic on this page, see “Example Tables for SAS SQL Procedure User’s Guide”. Download the ZIP file and extract the data files to a location that is accessible by SAS.

Houses

The Houses table is used in “Rewriting a GROUP BY Remerge Query in FedSQL” on page 1207.

```
data houses;
   input House $ x y;
   datalines;
house1 1 1
house2 3 3
house3 2 3
house4 7 7
;
```

Payroll

The Payroll table is used in “Rewriting a GROUP BY Remerge Query in FedSQL” on page 1207.

```
data payroll;
   input IdNumber $4. +3 Gender $1. +4 Jobcode $3. +9 Salary 5.
         +2 Birth date7. +2 Hired date7.;
   informat birth date7. hired date7.;
   format birth date7. hired date7.;
   datalines;
1919   M    TA2         34376  12SEP60  04JUN87
1653   F    ME2         35108  15OCT64  09AUG90
1400   M    ME1         29769  05NOV67  16OCT90
1350   F    FA3         32886  31AUG65  29JUL90
1401   M    TA3         38822  13DEC50  17NOV85
1499   M    ME3         43025  26APR54  07JUN80
1101   M    SCP         18723  06JUN62  01OCT90
1333   M    PT2         88606  30MAR61  10FEB81
1402   M    TA2         32615  17JAN63  02DEC90
1479   F    TA3         38785  22DEC68  05OCT89
1403   M    ME1         28072  28JAN69  21DEC91
1739   M    PT1         66517  25DEC64  27JAN91
1658   M    SCP         17943  08APR67  29FEB92
1428   F    PT1         68767  04APR60  16NOV91
1782   M    ME2         35345  04DEC70  22FEB92
1244   M    ME2         36925  31AUG63  17JAN88
1383   M    BCK         25823  25JAN68  20OCT92
1574   M    FA2         28572  27APR60  20DEC92
1789   M    SCP         18326  25JAN57  11APR78
1404   M    PT2         91376  24FEB53  01JAN80
1437   F    FA3         33104  20SEP60  31AUG84
```

1639	F	TA3	40260	26JUN57	28JAN84
1269	M	NA1	41690	03MAY72	28NOV92
1065	M	ME2	35090	26JAN44	07JAN87
1876	M	TA3	39675	20MAY58	27APR85
1037	F	TA1	28558	10APR64	13SEP92
1129	F	ME2	34929	08DEC61	17AUG91
1988	M	FA3	32217	30NOV59	18SEP84
1405	M	SCP	18056	05MAR66	26JAN92
1430	F	TA2	32925	28FEB62	27APR87
1983	F	FA3	33419	28FEB62	27APR87
1134	F	TA2	33462	05MAR69	21DEC88
1118	M	PT3	111379	16JAN44	18DEC80
1438	F	TA3	39223	15MAR65	18NOV87
1125	F	FA2	28888	08NOV68	11DEC87
1475	F	FA2	27787	15DEC61	13JUL90
1117	M	TA3	39771	05JUN63	13AUG92
1935	F	NA2	51081	28MAR54	16OCT81
1124	F	FA1	23177	10JUL58	01OCT90
1422	F	FA1	22454	04JUN64	06APR91
1616	F	TA2	34137	01MAR70	04JUN93
1406	M	ME2	35185	08MAR61	17FEB87
1120	M	ME1	28619	11SEP72	07OCT93
1094	M	FA1	22268	02APR70	17APR91
1389	M	BCK	25028	15JUL59	18AUG90
1905	M	PT1	65111	16APR72	29MAY92
1407	M	PT1	68096	23MAR69	18MAR90
1114	F	TA2	32928	18SEP69	27JUN87
1410	M	PT2	84685	03MAY67	07NOV86
1439	F	PT1	70736	06MAR64	10SEP90
1409	M	ME3	41551	19APR50	22OCT81
1408	M	TA2	34138	29MAR60	14OCT87
1121	M	ME1	29112	26SEP71	07DEC91
1991	F	TA1	27645	07MAY72	12DEC92
1102	M	TA2	34542	01OCT59	15APR91
1356	M	ME2	36869	26SEP57	22FEB83
1545	M	PT1	66130	12AUG59	29MAY90
1292	F	ME2	36691	28OCT64	02JUL89
1440	F	ME2	35757	27SEP62	09APR91
1368	M	FA2	27808	11JUN61	03NOV84
1369	M	TA2	33705	28DEC61	13MAR87
1411	M	FA2	27265	27MAY61	01DEC89
1113	F	FA1	22367	15JAN68	17OCT91
1704	M	BCK	25465	30AUG66	28JUN87
1900	M	ME2	35105	25MAY62	27OCT87
1126	F	TA3	40899	28MAY63	21NOV80
1677	M	BCK	26007	05NOV63	27MAR89
1441	F	FA2	27158	19NOV69	23MAR91
1421	M	TA2	33155	08JAN59	28FEB90
1119	M	TA1	26924	20JUN62	06SEP88
1834	M	BCK	26896	08FEB72	02JUL92
1777	M	PT3	109630	23SEP51	21JUN81
1663	M	BCK	26452	11JAN67	11AUG91
1106	M	PT2	89632	06NOV57	16AUG84
1103	F	FA1	23738	16FEB68	23JUL92
1477	M	FA2	28566	21MAR64	07MAR88
1476	F	TA2	34803	30MAY66	17MAR87

1379	M	ME3	42264	08AUG61	10JUN84
1104	M	SCP	17946	25APR63	10JUN91
1009	M	TA1	28880	02MAR59	26MAR92
1412	M	ME1	27799	18JUN56	05DEC91
1115	F	FA3	32699	22AUG60	29FEB80
1128	F	TA2	32777	23MAY65	20OCT90
1442	F	PT2	84536	05SEP66	12APR88
1417	M	NA2	52270	27JUN64	07MAR89
1478	M	PT2	84203	09AUG59	24OCT90
1673	M	BCK	25477	27FEB70	15JUL91
1839	F	NA1	43433	29NOV70	03JUL93
1347	M	TA3	40079	21SEP67	06SEP84
1423	F	ME2	35773	14MAY68	19AUG90
1200	F	ME1	27816	10JAN71	14AUG92
1970	F	FA1	22615	25SEP64	12MAR91
1521	M	ME3	41526	12APR63	13JUL88
1354	F	SCP	18335	29MAY71	16JUN92
1424	F	FA2	28978	04AUG69	11DEC89
1132	F	FA1	22413	30MAY72	22OCT93
1845	M	BCK	25996	20NOV59	22MAR80
1556	M	PT1	71349	22JUN64	11DEC91
1413	M	FA2	27435	16SEP65	02JAN90
1123	F	TA1	28407	31OCT72	05DEC92
1907	M	TA2	33329	15NOV60	06JUL87
1436	F	TA2	34475	11JUN64	12MAR87
1385	M	ME3	43900	16JAN62	01APR86
1432	F	ME2	35327	03NOV61	10FEB85
1111	M	NA1	40586	14JUL73	31OCT92
1116	F	FA1	22862	28SEP69	21MAR91
1352	M	NA2	53798	02DEC60	16OCT86
1555	F	FA2	27499	16MAR68	04JUL92
1038	F	TA1	26533	09NOV69	23NOV91
1420	M	ME3	43071	19FEB65	22JUL87
1561	M	TA2	34514	30NOV63	07OCT87
1434	F	FA2	28622	11JUL62	28OCT90
1414	M	FA1	23644	24MAR72	12APR92
1112	M	TA1	26905	29NOV64	07DEC92
1390	M	FA2	27761	19FEB65	23JUN91
1332	M	NA1	42178	17SEP70	04JUN91
1890	M	PT2	91908	20JUL51	25NOV79
1429	F	TA1	27939	28FEB60	07AUG92
1107	M	PT2	89977	09JUN54	10FEB79
1908	F	TA2	32995	10DEC69	23APR90
1830	F	PT2	84471	27MAY57	29JAN83
1882	M	ME3	41538	10JUL57	21NOV78
1050	M	ME2	35167	14JUL63	24AUG86
1425	F	FA1	23979	28DEC71	28FEB93
1928	M	PT2	89858	16SEP54	13JUL90
1480	F	TA3	39583	03SEP57	25MAR81
1100	M	BCK	25004	01DEC60	07MAY88
1995	F	ME1	28810	24AUG73	19SEP93
1135	F	FA2	27321	20SEP60	31MAR90
1415	M	FA2	28278	09MAR58	12FEB88
1076	M	PT1	66558	14OCT55	03OCT91
1426	F	TA2	32991	05DEC66	25JUN90
1564	F	SCP	18833	12APR62	01JUL92

```
1221   F    FA2         27896  22SEP67  04OCT91
1133   M    TA1         27701  13JUL66  12FEB92
1435   F    TA3         38808  12MAY59  08FEB80
1418   M    ME1         28005  29MAR57  06JAN92
1017   M    TA3         40858  28DEC57  16OCT81
1443   F    NA1         42274  17NOV68  29AUG91
1131   F    TA2         32575  26DEC71  19APR91
1427   F    TA2         34046  31OCT70  30JAN90
1036   F    TA3         39392  19MAY65  23OCT84
1130   F    FA1         23916  16MAY71  05JUN92
1127   F    TA2         33011  09NOV64  07DEC86
1433   F    FA3         32982  08JUL66  17JAN87
1431   F    FA3         33230  09JUN64  05APR88
1122   F    FA2         27956  01MAY63  27NOV88
1105   M    ME2         34805  01MAR62  13AUG90
;
```

Sales

The Sales table is used in “Rewriting a Query That Specifies the CALCULATED Keyword” on page 1211.

```
data Sales;
input Salesperson $ January February March;
datalines;
Smith 1000 650 800
Johnson 0 900 900
Reed 1200 700 850
Davis 1050 900 1000
Thompson 750 850 1000
Peterson 900 600 500
Jones 800 900 1200
Murphy 700 800 700
Garcia 400 1200 1150
;
```

Stores

The Stores table is used in “Rewriting a GROUP BY Remerge Query in FedSQL” on page 1207.

```
data stores;
  input Store $ x y;
  datalines;
store1 5 1
store2 5 3
store3 3 5
store4 7 5
;
```

Survey

The Survey table is used in “Using SAS Macros with FedSQL” on page 1217.

```
data survey;
  input id $ diet $ exer $ hours xwk educ;
  datalines;
1001 yes yes 1 3 1
1002 no  yes 1 4 2
1003 no  no  . . .n
1004 yes yes 2 3 .x
1005 no  yes 2 3 .x
1006 yes yes 2 4 .x
1007 no  yes .5 3 .
1008 no  no  . .  .
;
```

Appendix 7

FedSQL and the ANSI Standard

Compliance

FedSQL is core compliant with the 1999 ANSI Standard for SQL.[1]

1. International Organization for Standardization (ISO): Document ISO/IEC 9075–1:1999. Also available as American National Standards Institute (ANSI) Document ANSI 9075–1:1999.

FedSQL Enhancements

Reserved Words

For a complete list of reserved words in FedSQL, see “FedSQL Reserved Words” on page 71.

Column Modifiers

FedSQL supports the FORMAT=, INFORMAT=, and LABEL= modifiers in the HAVING clause of the CREATE TABLE statement. These modifiers control the format in which output data is displayed and labeled. For more information, see “How to Store, Change, Delete, and Use Stored Formats ” on page 78, “How to Specify Informats in FedSQL” on page 902, and the “HAVING Clause” on page 970.

The FedSQL SELECT statement preserves formats and labels in existing tables, and enables you to specify temporary formats with the PUT function. For more information, see “How to Format Output with the PUT Function” on page 79.

Functions

Aggregate Functions

FedSQL supports many more aggregate functions than are required by the ANSI Standard for SQL. For a complete list of aggregate functions that FedSQL supports, see “Aggregate Functions” on page 218.

Base SAS DATA Step Functions

FedSQL supports many of the functions that are available to the Base SAS DATA step. Functions that are not supported include the variable information functions and functions that work with arrays of data. Other FedSQL data sources support their own sets of functions.

For a complete list of functions that FedSQL supports, see “Overview of FedSQL Functions” on page 214.

Table Options

A table option specifies actions that enable you to perform operations on a table such as assigning or specifying passwords. A table option in the FedSQL performs much of the same functionality as a Base SAS data set option. For more information, see “Overview of Statement Table Options” on page 989.

SAS Missing Values and Null Values

FedSQL supports both SAS missing values and null values. Nonexistent data is represented by a SAS missing value in SAS data sets, SPD Engine data sets, and SPD Server tables. For all other data sources, nonexistent data is represented by an ANSI SQL null value. For more information, see “How FedSQL Processes Nulls and SAS Missing Values” on page 20.

DS2 Package Methods as Functions

You can invoke a DS2 package method as a function in a FedSQL SELECT statement. For more information, see “Using DS2 Packages in Expressions” on page 216.

Integrity Constraints

Integrity constraints are a set of data validation rules that you can specify to preserve the validity and consistency of your data. Integrity constraints that are specified in the CREATE TABLE statement are passed through to the data source. When a transaction modifies the table, the data source enforces integrity constraints. For more information, see the “CREATE TABLE Statement” on page 915.

FedSQL DICTIONARY Tables

FedSQL supports DICTIONARY tables that are similar to the DICTIONARY tables that are available in Base SAS. A FedSQL DICTIONARY table is a Read-only table that contains information about columns, tables, and catalogs, and statistics about a single table and its associated indexes. For more information, see “DICTIONARY Tables” on page 66.

FedSQL Limitations

Granting User Privileges

The GRANT statement, PRIVILEGES keyword, and authorization-identifier features of SQL are not supported. You might want to use operating environment-specific means of security instead.

Base SAS. SPD Engine, and SPD Server Identifiers and Naming Conventions

Table, column, and index names for SAS data sets, SPD Engine data sets, and SPD Server tables are limited to 32 characters. For more information, see “Identifiers” on page 18.

Appendix 8

Usage Notes

Usage Notes

Teradata: Enhancing Performance of FedSQL on Teradata

FedSQL queries that access system catalog tables (for example, the UPDATE statement, DROP TABLE statement, etc.) depend on joins of the system tables to obtain a result set from the Teradata database. In environments that create and drop a large number of database objects, performance suffers if the statistics about these system tables are not current. The following commands can be added to regular database maintenance processes to keep system table statistics up-to-date.

```
drop stats on dbc.tvm;
drop stats on dbc.owners;
drop stats on dbc.dbase;
drop stats on dbc.accessrights;
drop stats on dbc.tvfields;

collect stats on dbc.tvm column (tvmId);
collect stats on dbc.tvm INDEX ( DatabaseId ,TVMNameI );
collect stats on dbc.tvm column (DatabaseId );
collect stats on dbc.owners INDEX (ownerId);
collect stats on dbc.dbase INDEX ( DatabaseId );
collect stats on dbc.dbase column(JournalId);
collect stats on dbc.accessrights INDEX ( UserId ,DatabaseId );
collect stats on dbc.accessrights INDEX ( TVMId );
collect stats on dbc.accessrights column ( UserId ,TVMId);
collect stats on dbc.accessrights column (DatabaseId );
```

```
collect stats on dbc.tvfields column (DatabaseId);
collect stats on dbc.tvfields column (FieldId);
collect stats on dbc.tvfields column (tableId);


/* help stats displays what's there, this is optional */

help stats dbc.tvm;
help stats dbc.owners;
help stats dbc.dbase;
help stats dbc.accessrights;
help stats dbc.tvfields;



/* routine recollect, if stats have not been dropped */

collect stats on dbc.tvm ;

collect stats on dbc.owners ;
collect stats on dbc.dbase ;
collect stats on dbc.accessrights ;

collect stats on dbc.tvfields ;
```

Appendix 9

ICU License

ICU Licence: ICU 1.8.1–ICU 57 and ICU4J 1.3.1–ICU4J 57

DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

All trademarks and registered trademarks mentioned herein are the property of their respective owners.

Third-Party Software Licenses: ICU 1.8.1–ICU 57 and ICU4J 1.3.1–ICU4J 57

This section contains third-party software notices and/or additional terms for licensed third-party software components included within ICU libraries

1. Unicode Data Files and Software

COPYRIGHT AND PERMISSION NOTICE

Copyright © 1991-2015 Unicode, Inc. All rights reserved. Distributed under the Terms of Use in http://www.unicode.org/copyright.html.

Permission is hereby granted, free of charge, to any person obtaining a copy of the Unicode data files and any associated documentation (the "Data Files") or Unicode software and any associated documentation (the "Software") to deal in the Data Files or Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Data Files or Software, and to permit persons to whom the Data Files or Software are furnished to do so, provided that (a) the above copyright notice(s) and this permission notice appear with all copies of the Data Files or Software, (b) both the above copyright notice(s) and this permission notice appear in associated documentation, and (c) there is clear notice in each modified Data File or in the Software as well as in the documentation associated with the Data File(s) or Software that the data or software has been modified.

THE DATA FILES AND SOFTWARE ARE PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING

FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THE DATA FILES OR SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in these Data Files or Software without prior written authorization of the copyright holder.

2. Chinese/Japanese Word Break Dictionary Data (cjdict.txt)

```
#  The Google Chrome software developed by Google is licensed under
#  the BSD license. Other software included in this distribution is provided
#  under other licenses, as set forth below.
#
#    The BSD License
#    http://opensource.org/licenses/bsd-license.php
#    Copyright (C) 2006-2008, Google Inc.
#
#    All rights reserved.
#
#    Redistribution and use in source and binary forms, with or without
#  modification, are permitted provided that the following conditions are met:
#
#    Redistributions of source code must retain the above copyright notice, this
#  list of conditions and the following disclaimer.
#    Redistributions in binary form must reproduce the above copyright notice,
#  this list of conditions and the following disclaimer in the documentation
#  and/or other materials provided with the distribution.
#    Neither the name of Google Inc. nor the names of its contributors may be
#  used to endorse or promote products derived from this software without
#  specific prior written permission.
#
#
#  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
#  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
#  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
#  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
#  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
#  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
#  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
#  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
#  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
#  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
#  THE POSSIBILITY OF SUCH DAMAGE.
#
#
#    The word list in cjdict.txt are generated by combining three word lists
#  listed below with further processing for compound word breaking. The
#  frequency is generated with an iterative training against Google
#  web corpora.
#
```

```
#    * Libtabe (Chinese)
#      - https://sourceforge.net/project/?group_id=1519
#      - Its license terms and conditions are shown below.
#
#    * IPADIC (Japanese)
#      - http://chasen.aist-nara.ac.jp/chasen/distribution.html
#      - Its license terms and conditions are shown below.
#
#    ---------COPYING.libtabe ---- BEGIN--------------------
#
#    /*
#     * Copyrighy (c) 1999 TaBE Project.
#     * Copyright (c) 1999 Pai-Hsiang Hsiao.
#     * All rights reserved.
#     *
#     * Redistribution and use in source and binary forms, with or without
#     * modification, are permitted provided that the following conditions
#     * are met:
#     *
#     * . Redistributions of source code must retain the above copyright
#     *   notice, this list of conditions and the following disclaimer.
#     * . Redistributions in binary form must reproduce the above copyright
#     *   notice, this list of conditions and the following disclaimer in
#     *   the documentation and/or other materials provided with the
#     *   distribution.
#     * . Neither the name of the TaBE Project nor the names of its
#     *   contributors may be used to endorse or promote products derived
#     *   from this software without specific prior written permission.
#     *
#     * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
#     * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
#     * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
#     * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
#     * REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
#     * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
#     * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
#     * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
#     * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
#     * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
#     * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
#     * OF THE POSSIBILITY OF SUCH DAMAGE.
#     */
#
#    /*
#     * Copyright (c) 1999 Computer Systems and Communication Lab,
#     *                    Institute of Information Science, Academia Sinica.
#     * All rights reserved.
#     *
#     * Redistribution and use in source and binary forms, with or without
#     * modification, are permitted provided that the following conditions
#     * are met:
#     *
#     * . Redistributions of source code must retain the above copyright
#     *   notice, this list of conditions and the following disclaimer.
#     * . Redistributions in binary form must reproduce the above copyright
#     *   notice, this list of conditions and the following disclaimer in
```

```
#     *   the documentation and/or other materials provided with the
#     *   distribution.
#     * . Neither the name of the Computer Systems and Communication Lab
#     *   nor the names of its contributors may be used to endorse or
#     *   promote products derived from this software without specific
#     *   prior written permission.
#     *
#     * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
#     * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
#     * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
#     * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
#     * REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
#     * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
#     * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
#     * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
#     * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
#     * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
#     * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
#     * OF THE POSSIBILITY OF SUCH DAMAGE.
#     */
#
#    Copyright 1996 Chih-Hao Tsai @ Beckman Institute, University of Illinois
#    c-tsai4@uiuc.edu  http://casper.beckman.uiuc.edu/~c-tsai4
#
#    ---------------COPYING.libtabe-----END-----------------------------------
#
#
#    ---------------COPYING.ipadic-----BEGIN-----------------------------------
#
#    Copyright 2000, 2001, 2002, 2003 Nara Institute of Science
#    and Technology.  All Rights Reserved.
#
#    Use, reproduction, and distribution of this software is permitted.
#    Any copy of this software, whether in its original form or modified,
#    must include both the above copyright notice and the following
#    paragraphs.
#
#    Nara Institute of Science and Technology (NAIST),
#    the copyright holders, disclaims all warranties with regard to this
#    software, including all implied warranties of merchantability and
#    fitness, in no event shall NAIST be liable for
#    any special, indirect or consequential damages or any damages
#    whatsoever resulting from loss of use, data or profits, whether in an
#    action of contract, negligence or other tortuous action, arising out
#    of or in connection with the use or performance of this software.
#
#    A large portion of the dictionary entries
#    originate from ICOT Free Software.  The following conditions for ICOT
#    Free Software applies to the current dictionary as well.
#
#    Each User may also freely distribute the Program, whether in its
#    original form or modified, to any third party or parties, PROVIDED
#    that the provisions of Section 3 ("NO WARRANTY") will ALWAYS appear
#    on, or be attached to, the Program, which is distributed substantially
#    in the same form as set out herein and that such intended
#    distribution, if actually made, will neither violate or otherwise
```

```
#    contravene any of the laws and regulations of the countries having
#    jurisdiction over the User or the intended distribution itself.
#
#    NO WARRANTY
#
#    The program was produced on an experimental basis in the course of the
#    research and development conducted during the project and is provided
#    to users as so produced on an experimental basis.  Accordingly, the
#    program is provided without any warranty whatsoever, whether express,
#    implied, statutory or otherwise.  The term "warranty" used herein
#    includes, but is not limited to, any warranty of the quality,
#    performance, merchantability and fitness for a particular purpose of
#    the program and the nonexistence of any infringement or violation of
#    any right of any third party.
#
#    Each user of the program will agree and understand, and be deemed to
#    have agreed and understood, that there is no warranty whatsoever for
#    the program and, accordingly, the entire risk arising from or
#    otherwise connected with the program is assumed by the user.
#
#    Therefore, neither ICOT, the copyright holder, or any other
#    organization that participated in or was otherwise related to the
#    development of the program and their respective officials, directors,
#    officers and other employees shall be held liable for any and all
#    damages, including, without limitation, general, special, incidental
#    and consequential damages, arising out of or otherwise in connection
#    with the use or inability to use the program or any product, material
#    or result produced or otherwise obtained by using the program,
#    regardless of whether they have been advised of, or otherwise had
#    knowledge of, the possibility of such damages at any time during the
#    project or thereafter.  Each user will be deemed to have agreed to the
#    foregoing by his or her commencement of use of the program.  The term
#    "use" as used herein includes, but is not limited to, the use,
#    modification, copying and distribution of the program and the
#    production of secondary products from the program.
#
#    In the case where the program, whether in its original form or
#    modified, was distributed or delivered to or received by a user from
#    any person, organization or entity other than ICOT, unless it makes or
#    grants independently of ICOT any specific warranty to the user in
#    writing, such person, organization or entity, will also be exempted
#    from and not be held liable to the user for any such damages as noted
#    above as far as the program is concerned.
#
#    ---------------COPYING.ipadic-----END-----------------------------------
```

3. Lao Word Break Dictionary Data (laodict.txt)

```
#    Copyright (c) 2013 International Business Machines Corporation
#    and others. All Rights Reserved.
#
#    Project:    http://code.google.com/p/lao-dictionary/
#    Dictionary: http://lao-dictionary.googlecode.com/git/Lao-Dictionary.txt
```

```
#    License:    http://lao-dictionary.googlecode.com/git/Lao-Dictionary-LICENSE.txt
#                (copied below)
#
#    This file is derived from the above dictionary, with slight modifications.
#    --------------------------------------------------------------------------------
#    Copyright (C) 2013 Brian Eugene Wilson, Robert Martin Campbell.
#    All rights reserved.
#
#    Redistribution and use in source and binary forms, with or without modification,
#    are permitted provided that the following conditions are met:
#
#        Redistributions of source code must retain the above copyright notice, this
#        list of conditions and the following disclaimer. Redistributions in binary
#        form must reproduce the above copyright notice, this list of conditions and
#        the following disclaimer in the documentation and/or other materials
#        provided with the distribution.
#
#    THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
#    ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
#    WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
#    DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
#    ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
#    (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
#    LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
#    ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
#    (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
#    SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
#    --------------------------------------------------------------------------------
```

4. Burmese Word Break Dictionary Data (burmesedict.txt)

```
#    Copyright (c) 2014 International Business Machines Corporation
#    and others. All Rights Reserved.
#
#    This list is part of a project hosted at:
#       github.com/kanyawtech/myanmar-karen-word-lists
#
#    --------------------------------------------------------------------------------
#    Copyright (C) 2013 LeRoy Benjamin Sharon
#    All rights reserved.
#
#    Redistribution and use in source and binary forms, with or without
#    modification, are permitted provided that the following conditions
# are met: Redistributions of source code must retain the above
#    copyright notice, this list of conditions and the following
#    disclaimer.  Redistributions in binary form must reproduce the
# above copyright notice, this list of conditions and the following
#    disclaimer in the documentation and/or other materials provided
# with the distribution
#
#        Neither the name Myanmar Karen Word Lists, nor the names of its
```

```
#        contributors may be used to endorse or promote products derived
#        from this software without specific prior written permission.
#
#    THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
#    ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
#    WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
#    DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
#    ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
#    (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
#    LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
#    ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
#    (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
#    SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
#    ---------------------------------------------------------------------------
```

3. Time Zone Database

ICU uses the public domain data and code derived from Time Zone Database for its time zone support. The ownership of the TZ database is explained in BCP 175: Procedure for Maintaining the Time Zone Database section 7.

```
7.  Database Ownership

   The TZ database itself is not an IETF Contribution or an IETF
   document.  Rather it is a pre-existing and regularly updated work
   that is in the public domain, and is intended to remain in the
public
   domain.  Therefore, BCPs 78 [RFC5378] and 79 [RFC3979] do not apply
   to the TZ Database or contributions that individuals make to it.
   Should any claims be made and substantiated against the TZ Database,
   the organization that is providing the IANA Considerations defined
in
   this RFC, under the memorandum of understanding with the IETF,
   currently ICANN, may act in accordance with all competent court
   orders.  No ownership claims will be made by ICANN or the IETF Trust
   on the database or the code.  Any person making a contribution to
the
   database or code waives all rights to future claims in that
   contribution or in the TZ Database.
```

Unicode, Inc. License Agreement - Data Files and Software: ICU 58 and Later

See Terms of Use for definitions of Unicode Inc.'s Data Files and Software.

NOTICE TO USER: Carefully read the following legal agreement. BY DOWNLOADING, INSTALLING, COPYING OR OTHERWISE USING UNICODE