SAS® FedSQL Language Reference for SAS® Cloud Analytic Services 3.1

SAS® Documentation

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2016. *SAS® FedSQL Language Reference for SAS® Cloud Analytic Services 3.1*. Cary, NC: SAS Institute Inc.

SAS® FedSQL Language Reference for SAS® Cloud Analytic Services 3.1

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

September 2016

3.1-P1:casfedsql

Contents

Part 1

Concepts

Chapter 1

Introduction to SAS FedSQL for CAS

Introduction to the SAS FedSQL Language for CAS

SAS FedSQL is a SAS proprietary implementation of ANSI SQL:1999 core standard. It provides support for industry-standard data types and other ANSI 1999 core compliance features and proprietary extensions.

For data that has been loaded into SAS Cloud Analytic Services, FedSQL provides a scalable, threaded, high-performance way to query data. When possible, FedSQL queries are optimized with multi-threaded algorithms in order to resolve large-scale operations.

In its initial release for SAS Cloud Analytic Services, FedSQL can be used to query in-memory CAS tables and to create in-memory CAS output tables from CAS input tables. The input data must previously have been loaded into the SAS Cloud Analytic Services server. FedSQL provides a way to join data using industry-standard query expressions and SQL expressions.

The FEDSQL procedure enables you to submit FedSQL language statements to CAS from a SAS Studio session. For more information about PROC FEDSQL, see *SAS Viya Data Management and Utility Procedures Guide*. In addition, you can use the FedSQL.execDirect action to submit FedSQL language statements to CAS. For more information about the FedSQL.execDirect action, see *SAS Cloud Analytic Services: System Programming Guide*.

Running FedSQL Programs in CAS

You can submit FedSQL programs to a CAS server in several ways:

- In SAS Studio using the FEDSQL procedure. The FEDSQL procedure can be used to run FedSQL statements in SAS Viya or in the CAS server. For more information, see "FEDSQL" in *SAS Viya Data Management and Utility Procedures Guide*.
- In SAS Studio using the FedSQL.execDirect action. The FedSQL.execDirect action is used in conjunction with the CAS procedure. For more information about the CAS procedure, see *SAS Cloud Analytic Services: CAS Procedure Programming Guide and Reference*. For more information about the FedSQL.execDirect action, see *SAS Cloud Analytic Services: System Programming Guide*.
- From a Python or Lua program using the FedSQL.execDirect action. For more information, see *SAS Cloud Analytic Services: Getting Started with Python* or *SAS Cloud Analytic Services: Getting Started with Lua*.

Note: Unless you are using Python or Lua, it is recommended that you use PROC FEDSQL to submit FedSQL statements to the CAS server.

Supported Data Sources

In its initial release on SAS Cloud Analytic Services, FedSQL supports queries on SAS Cloud Analytic Services tables only.

Intended Audience

The information in this document is intended for the following users who perform these roles:

- Application developers who write the client applications that 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 want or need to take advantage of the features of the FedSQL language.

Syntax Conventions for the FedSQL Language

Typographical Conventions

Type styles have special meanings when used in the documentation of the FedSQL language syntax.

`UPPERCASE BOLD`
identifies FedSQL keywords such the names of statements and functions (for example, PUT).

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

italic
identifies arguments or values that you supply. Items in italic represent user-supplied values that are nonliteral arguments (for example, AVG=*expression*).

`monospace`
identifies examples of SAS code.

Syntax Conventions

SAS FedSQL Language Reference for SAS Cloud Analytic Services uses the Backus-Naur Form (BNF), specifically the same syntax notation 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 FedSQL language reference documentation is in how optional syntax arguments are displayed. In traditional SAS syntax, angle brackets (< >) are used to denote optional syntax. In FedSQL language 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 language 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 square brackets unless they are preceded by a backward slash (\), which denotes that they are literal.

{ }
Braces distinguish required multi-word arguments. Do not enter 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 that are 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 the next character is a literal.

Chapter 2

Fundamental Concepts

Statements

FedSQL supports the following statements in CAS:

- CREATE TABLE, with the AS query expression
- DROP TABLE
- SELECT

Data Types

A data type is an attribute of every column in a table that specifies the type of data the column stores. FedSQL supports a broad range of data types. However, CAS tables support only three data types in the initial release. Because FedSQL does not support joining other data sources with CAS tables, all interactions use the following data types.

Table 2.1 *FedSQL Data Types Supported in CAS*

Data Type	Description
CHAR(*n*)	Stores a fixed-length character string, where *n* is the maximum number of characters to store. The maximum number of characters is required to store each value regardless of the actual size of the value. If char(10) is specified and the character string is only five characters long, the value is right-padded with spaces.
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. For SAS Cloud Analytic Services, this is a 64-bit double precision, floating-point number.
VARCHAR(*n*)	Stores a varying-length character string.

Each date, time, and timestamp value is stored as a DOUBLE in CAS tables.

CAS tables use the UTF-8 character set.

Identifiers

Overview of Identifiers

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

Regular Identifiers

Regular identifiers are the type of identifiers that you see in most programming languages. They are not case-sensitive. Only certain characters are allowed in 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. For example, phone_num1 is the same as Phone_Num1 and PHONE_NUM1.

Delimited Identifiers

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

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. The identifier “phone_num” is not equivalent to “Phone_Num” or “PHONE_NUM”. The delimited identifier “PHONE_NUM” is equivalent to the regular identifier “phone_num”.

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 10.

Support for Non-Latin Characters

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

Formats

A format is an instruction that SAS languages such as the DATA step, DS2, and FedSQL use to write data values. SAS programs 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.

FedSQL preserves formats that exist on CAS input tables in CAS output tables that it creates. It also enables you to specify temporary formats on columns in the SELECT statement with the PUT function. For more information, see Chapter 6, "FedSQL Formats," on page 63.

Handling of Nonexistent Data

Nonexistent data in CAS tables is represented by a SAS missing value. The SAS missing value indicator is a dot (or period). This handling is different from ANSI SQL handling of nulls and missing values. In ANSI SQL, nulls and missing values have no data value. That is, nulls are treated as unknown values. In CAS, they are treated as known values.

The use of missing values has implications for query processing. Consider these differences in filtering data (for example, in a WHERE clause, a HAVING clause, or an outer join ON clause). FedSQL interprets null values as known values, but ANSI mode interprets null values as unknown values.

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 9.

Table 2.2 *FedSQL Reserved Words A - D*

A

ABORT
ABSOLUTE
ACCESS
ACTION
ADD
AFTER
AGGREGATE
ALL
ALLOCATE
ALTER
ANALYSE
ANALYZE
AND
ANY
ARE
ARRAY
AS
ASC
ASENSITIVE
ASSERTION
ASSIGNMENT
ASYMMETRIC
AT
ATOMIC
AUTORIZATION

B

BACKWARD
BEFORE
BEGIN
BETWEEN
BIGINT
BINARY
BIT
BLOB
BOOLEAN
BOTH
BY

C

CACHE
CALL
CALLED
CARDINALITY
CASCADE
CASCADED
CASE
CAST
CHAIN
CHAR
CHAR_LENGTH
CHARACTER
CHARACTER_LENGTH
CHARACTERISTICS
CHECK
CHECKPOINT
CLASS
CLOB
CLOSE
CLUSTER
COALESCE
COLLATE
COLLECT
COLUMN
COMMENT
COMMIT
COMMITTED
CONDITION
CONNECT
CONSTRAINT
CONSTRAINTS
CONVERSION
CONVERT
COPY
CORR
CORRESPONDING
COVAR_POP
COVAR_SAMP
CREATE
CREATEDB
CREATEUSER
CROSS
CUBE
CUME_DIST
CURRENT
CURRENT_DATE
CURRENT_DEFAULT_TRANSFORM_GROUP
CURRENT_PATH
CURRENT_ROLE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TRANSFORM_GROUP_FOR_TYPE
CURRENT_USER
CURSOR
CYCLE

D

DATABASE
DAY
DEALLOCATE
DEC
DECIMAL
DECLARE
DEFAULT
DEFAULTS
DEFERRABLE
DEFERRED
DEFINER
DELETE
DELIMITER
DELIMITERS
DENSE_RANK
DEREF
DESC
DESCRIBE
DETERMINISTIC
DISCONNECT
DISTINCT
DO
DOMAIN
DOUBLE
DROP
DYNAMIC

Table 2.3 *FedSQL Reserved Words E - O*

E
EACH
ELEMENT
ELSE
ENCODING
ENCRYPTED
END
END-EXEC
ESCAPE
EVERY
EXCEPT
EXCLUDING
EXCLUSIVE
EXEC
EXECUTE
EXISTS
EXPLAIN
EXTERNAL
EXTRACT

F
FALSE
FETCH
FILTER
FIRST
FLOAT
FOR
FORCE
FOREIGN
FORWARD
FREE
FREEZE
FROM
FULL
FUNCTION
FUSION

G
GET
GLOBAL
GRANT
GROUP
GROUPING

H
HANDLER
HAVING
HOLD

I
ILIKE
IMMEDIATE
IMMUTABLE
IMPLICIT
IN
INCLUDING
INCREMENT
INDEX
INDICATOR
INHERITS
INITIALLY
INNER
INOUT
INPUT
INSENSITIVE
INSERT
INSTEAD
INT
INTEGER
INTERSECT
INTERSECTION
INTERVAL
INTO
INVOKER
IS
ISNULL
ISOLATION

J
JOIN

K
KEY

L
LANCOMPILER
LANGUAGE
LARGE
LAST
LATERAL
LEADING
LEFT
LEVEL
LIKE
LIMIT
LISTEN
LOAD
LOCAL
LOCALTIME
LOCALTIMESTAMP
LOCATION
LOCK

M
MATCH
MAXVALUE
MEMBER
MERGE
METHOD
MINUTE
MINVALUE
MODE
MODIFIES
MODULE
MONTH
MOVE
MULTISET

N
NAMES
NATIONAL
NATURAL
NCHAR
NCLOB
NEW
NEXT
NO
NOCREATEDB
NOCREATEUSER
NONE
NORMALIZE
NOT
NOTHING
NOTIFY
NOTNULL
NULL
NULLIF
NUMERIC

O
OF
OFF
OFFSET
OIDS
OLD
ON
ONLY
OPEN
OPERATOR
OPTION
OR
ORDER
OUT
OUTER
OVER
OVERLAPS
OVERLAY
OWNER

Table 2.4 *FedSQL Reserved Words P - Z*

P

PARAMETER
PARTIAL
PARTITION
PASSWORD
PATH
PENDANT
PERCENT_RANK
PERCENTILE_CONT
PERCENTILE_DESC
PLACING
POSITION
PRECISION
PREPARE
PRESERVE
PRIMARY
PRIOR
PRIVILEGES
PROCEDURAL
PROCEDURE

R

RANK
READ
READS
REAL
RECHECK
RECURSIVE
REF
REFERENCES
REFERENCING
REGR_AVGX
REGR_AVGY
REGR_COUNT
REGR_INTERCEPT
REGR_R2
REGR_SLOPE
REGR_SXX
REGR_SXY
REGR_SYY
REINDEX
RELATIVE
RENAME
REPLACE
RESET
RESTART
RESTRICT
RESULT
RETURN
RETURNS
REVOKE
RIGHT
ROLLBACK
ROLLUP
ROW
ROWS
ROW_NUMBER
RULE

S

SCHEMA
SCOPE
SCROLL
SEARCH
SECOND
SECURITY
SELECT
SENSITIVE
SPECIFIC
SPECIFICTYPE
SEQUENCE
SERIALIZABLE
SESSION
SESSION_USER
SET
SETOF
SHARE
SHOW
SIMILAR
SIMPLE
SMALLINT
SOME
SQLEXCEPTION
SQLSTATE
SQLWARNING
STABLE
START
STATEMENT
STATIC
STATISTICS
STDDEV_POP
STDDEV_SAMP
STDIN
STDOUT
STORAGE
STRICT
SUBMULTISET
SUBSTRING
SYMMETRIC
SYSID
SYSTEM
SYSTEM_USER

T

TABLE
TABLESAMPLE
TEMP
TEMPLATE
TEMPORARY
THEN
TIME
TIMESTAMP
TIMEZONE_HOUR
TIMEZONE_MINUTE
TO
TOAST
TRAILING
TRANSACTION
TRANSLATE
TRANSLATION
TREAT
TRIGGER
TRIM
TRUE
TRUNCATE
TRUSTED
TYPE

U

UESCAPE
UNENCRYPTED
UNION
UNIQUE
UNKNOWN
UNLISTEN
UNNEST
UNTIL
UPDATE
USAGE
USER
USING

V

VACUUM
VALID
VALIDATOR
VALUE
VALUES
VARCHAR
VARYING
VAR_POP
VAR_SAMP
VERBOSE
VERSION
VIEW
VOLATILE

W

WHEN
WHENEVER
WHERE
WIDTH_BUCKET
WINDOW
WITH
WITHIN
WITHOUT
WORK
WRITE

Y

YEAR

Z

ZONE

Dictionary Tables

The FedSQL language does not support dictionary queries in CAS. You can obtain column information for writing queries in the following ways:

- From SAS Viya:
  - use the CASUTIL procedure with the CONTENTS statement.
  - use the CAS procedure with the DESCRIBE statement.
  - use PROC CONTENTS.
- From Python and Lua, use the fileInfo action from the Table action set.

Chapter 3

Joining Data with FedSQL

Overview of Joins

A join operation is a query that combines data from two or more tables 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.

Most joins are of two tables. However, you can join more than two tables. When a join operation is requested on 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.

FedSQL supports 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.

simple join
: multiple tables, separated by commas, are listed in the FROM clause of a SELECT statement. The join can include all or specified columns from the input tables. There is no join condition.

equijoin
: a simple join that is subset with a WHERE clause. The join condition is an equality comparison.

cross join
: a join of two tables requested by inserting the keywords CROSS JOIN between the table names in the FROM clause. A cross join obtains similar results as a simple join, except that it can be subset with a WHERE clause. You cannot use an ON clause. A CROSS JOIN is as referred to as a *relational join*. You can also specify a WHERE clause.

qualified join
: a join of two tables requested by inserting the keyword JOIN between the table names in the FROM clause. The returned rows are filtered based on the column specified in an ON clause or USING clause. You can use a WHERE clause to further subset the query results.

natural join
: a join of two tables requested by inserting the keywords NATURAL JOIN between the table names in the FROM clause. The natural join selects rows from two tables that have equal values in columns that share the same name and data type. You can specify a subset of the columns from the input tables.

inner join
: a join of two tables requested by inserting the keywords INNER JOIN between the table names in the FROM clause. An inner join returns a result set that includes all rows from the first table that match rows from the second table. Inner joins return only those rows that satisfy the join condition. Unmatched rows from both tables are discarded.

outer join
: a join of two tables requested by inserting the keywords OUTER JOIN between the table names in the FROM clause. 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.

- A left outer join is requested with the syntax LEFT [OUTER]. A left outer join returns a result set that includes all rows that satisfy the join condition and rows from the left (first) table that do not match the join condition.
- A right outer join is requested with the syntax RIGHT [OUTER]. A right outer join returns a result set that includes all rows that satisfy the join condition and rows from the right (second) table that do not match the join condition.
- A full outer join is requested with the syntax FULL [OUTER]. A full outer join returns all matching and unmatching rows from the left and right table.

Join Examples

The examples in this section use the tables Customers, Products, and Sales, which are described in Appendix 1, "Tables Used in Examples," on page 351. The first two examples illustrate typical join operations. The remaining examples illustrate how the various join types can be used to manipulate the same data. Some join types (for example, the equijoin and cross join with a WHERE clause) accomplish the same result, with slightly different syntax.

Typical Two-Table Join

Program

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

Here is the output from the SELECT statement:

Output 3.1 *Result Set from Join of Tables Products and Sales*

PRODUCT	TOTALS
Corn	$555,789
Wheat	$781,183
Wheat	$2,789,654
Rice	$189,400
Barley	$899,453

Key Ideas

- Most join operations contain at least one join condition, which is either in the FROM clause or in a WHERE clause. This example specifies a WHERE clause.
- The query selects a column from each input table (Product from table "Products" on page 354 and Totals from table "Sales" on page 355) and merges the content based on the value of a third column that is common to both tables, Prodid.
- The table columns are identified by using a two-part name in the form *table-name.column-name*.
- Because the input tables are specified in a comma-separated list, this query is considered a simple join.

Typical Three-Table Join

Program

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

Here is the output from the SELECT statement:

Output 3.2 *Result Set from Join of Tables Products, Sales, and Customers*

PRODUCT	TOTALS	CITY
Corn	$555,789	Nagasaki
Wheat	$781,183	Tokyo
Wheat	$2,789,654	Little Rock
Rice	$189,400	Boulder
Barley	$899,453	Buenos Aires

Key Ideas

- This FedSQL SELECT statement specifies a column from each of three tables: Product from table "Products" on page 354, Totals from table "Sales" on page 355, and City from table "Customers" on page 352 to form the result set.
- 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. This 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.
- Because of the syntax (comma-separated list of input tables), this query is also considered to be a simple join.

Simple Join Including All Columns

Program

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

Here is the output from the SELECT statement:

Output 3.3 *Simple Join of Tables Products and Sales*

PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
3421	Wheat	3421	4	$781,183	Japan
3421	Wheat	3234	1	$189,400	United States
3421	Wheat	3975	5	$899,453	Argentina
3421	Wheat	1424	3	$555,789	Japan
3421	Wheat	3421	2	$2,789,654	United States
3975	Barley	3975	5	$899,453	Argentina
3975	Barley	3421	4	$781,183	Japan
3975	Barley	3234	1	$189,400	United States
3975	Barley	1424	3	$555,789	Japan
3975	Barley	3421	2	$2,789,654	United States
1424	Corn	1424	3	$555,789	Japan
1424	Corn	3234	1	$189,400	United States
1424	Corn	3421	4	$781,183	Japan
1424	Corn	3975	5	$899,453	Argentina
1424	Corn	3421	2	$2,789,654	United States
3422	Oat	3421	2	$2,789,654	United States
3422	Oat	3421	4	$781,183	Japan
3422	Oat	3234	1	$189,400	United States
3422	Oat	3975	5	$899,453	Argentina
3422	Oat	1424	3	$555,789	Japan
3234	Rice	3234	1	$189,400	United States
3234	Rice	3421	4	$781,183	Japan
3234	Rice	3975	5	$899,453	Argentina

Key Ideas

- This is the simplest form of the simple join. The FedSQL SELECT statement specifies to merge all of the columns from two tables, "Products" on page 354 and "Sales" on page 355, and display the results as if they were a single table. The asterisk specifies that all columns should be included.
- 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 is referred to as a *Cartesian join*. The result is a large, basically meaningless result set. Typically, you want to filter the results with a WHERE clause or JOIN expression.

Equijoin Including All Columns

Program

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

Here is the output from the SELECT statement:

Output 3.4 *Equijoin of All Columns*

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

Key Ideas

- 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" on page 354 and "Sales" on page 355 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. If you were to 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.

Simple Cross Join

Program

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

Here is the output from the SELECT statement:

Output 3.5 *Cross Join of Two Tables*

PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
1424	Corn	1424	3	$555,789	Japan
1424	Corn	3234	1	$189,400	United States
1424	Corn	3421	4	$781,183	Japan
1424	Corn	3421	2	$2,789,654	United States
1424	Corn	3975	5	$899,453	Argentina
3421	Wheat	3421	4	$781,183	Japan
3421	Wheat	3234	1	$189,400	United States
3421	Wheat	1424	3	$555,789	Japan
3421	Wheat	3421	2	$2,789,654	United States
3421	Wheat	3975	5	$899,453	Argentina
3234	Rice	3234	1	$189,400	United States
3234	Rice	1424	3	$555,789	Japan
3234	Rice	3421	4	$781,183	Japan
3234	Rice	3421	2	$2,789,654	United States
3234	Rice	3975	5	$899,453	Argentina
3422	Oat	3421	2	$2,789,654	United States
3422	Oat	3234	1	$189,400	United States
3422	Oat	1424	3	$555,789	Japan
3422	Oat	3421	4	$781,183	Japan
3422	Oat	3975	5	$899,453	Argentina
3975	Barley	3975	5	$899,453	Argentina
3975	Barley	3234	1	$189,400	United States
3975	Barley	1424	3	$555,789	Japan
3975	Barley	3421	4	$781,183	Japan
3975	Barley	3421	2	$2,789,654	United States

Key Ideas

- A cross join is requested with the syntax CROSS JOIN. A cross join is a relational join that results in a Cartesian product of two tables.
- This cross join example selects all columns and all rows from the tables Products and Sales, and it produces the same results as a simple join of all columns of the two tables.

Cross Join with Specified Columns and a WHERE Clause

Program

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

Here is the output from the SELECT statement:

Output 3.6 *Result Set from Cross Join with a WHERE Clause*

PRODID	PRODUCT	TOTALS
1424	Corn	$555,789
3421	Wheat	$781,183
3421	Wheat	$2,789,654
3234	Rice	$189,400
3975	Barley	$899,453

Key Ideas

- A cross join can be subset with a WHERE clause, but you cannot use an ON clause.
- This cross join example selects the columns Prodid and Product from the tables "Products" on page 354 and column Totals from "Sales" on page 355. 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.

Qualified Join with an ON Clause

Program

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

Here is the output from the SELECT statement:

Output 3.7 *Results of Qualified Join with an ON Clause*

PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
1424	Corn	3234	1	$189,400	United States
1424	Corn	3421	2	$2,789,654	United States
3421	Wheat	3234	1	$189,400	United States
3421	Wheat	3421	2	$2,789,654	United States
3422	Oat	3421	2	$2,789,654	United States
3422	Oat	3234	1	$189,400	United States
3234	Rice	3234	1	$189,400	United States
3234	Rice	3421	2	$2,789,654	United States
3975	Barley	3234	1	$189,400	United States
3975	Barley	3421	2	$2,789,654	United States

Key Ideas

- A qualified join requests a join of two tables by inserting the keyword JOIN between the table names in the FROM clause. The returned rows are filtered based on the column specified in an ON clause or USING clause. 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.

- This example uses an ON clause to specify 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.
- This qualified join example selects all columns from the tables "Products" on page 354 and "Sales" on page 355. 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. The filter column name and value are enclosed within parentheses.
- A qualified join can be an inner join or an outer join. These joins are requested with the syntax INNER or OUTER. If the join type specification is omitted, then an inner join is implied.

Related Information

- "Inner and Outer Join Types" on page 26
- "Example of a Left Outer Qualified Join" on page 27
- "Example of Right Outer Qualified Join" on page 28
- "Example of a Full Outer Qualified Join" on page 29

Qualified Join with a USING Clause

Program

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

Here is the output from the SELECT statement:

Output 3.8 *Result Set of Qualified Join with a USING Clause*

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

Key Ideas

- A qualified join with a 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.
- This qualified join example selects all columns from the tables "Products" on page 354 and "Sales" on page 355. The returned rows are filtered by selecting the values

that match for the column Prodid, which exists in both tables. The column Prodid is enclosed within parentheses.

Note that unlike an equijoin and a cross join, the column Prodid is not duplicated in the result set.

- 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.

Related Information

- "Inner and Outer Join Types" on page 26

Qualified Join with an ON Clause and a WHERE Clause

Program

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

Here is the output from the SELECT statement.

Output 3.9 *Result Set from Qualified Join with an ON Clause and WHERE Clause*

PRODID	PRODUCT	TOTALS
3234	Rice	$189,400
3234	Rice	$2,789,654

Key Ideas

This qualified join example selects columns Prodid and Product from table "Products" on page 354 and column Totals from table "Sales" on page 355. The returned rows are filtered based on the column Country from table Sales where the value equals United States. The returned rows are further subset where the value for Product equals Rice.

Related Information

- "Inner and Outer Join Types" on page 26

Natural Join

Program

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

Here is the output from the SELECT statement:

Output 3.10 *Result Set of Natural Join of All Columns in Tables Products and Sales*

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

Key Ideas

- 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.
- This natural join example selects all columns from the tables "Products" on page 354 and "Sales" on page 355. 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.

Related Information

- "Inner and Outer Join Types" on page 26

Natural Join with a WHERE Clause

Program

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

Here is the output from the SELECT statement:

Output 3.11 *Result Set of Natural Join with a WHERE Clause*

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

Key Ideas

This natural join example selects columns City and Totals from the tables "Sales" on page 355 and "Customers" on page 352. 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.

Related Information

- "Inner and Outer Join Types" on page 26

Inner and Outer Join Types

Understanding the 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.

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.

Example of an Inner Join

Program

```
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';
```

Key Ideas

- 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.

Example of a Left Outer Qualified Join

Program

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

Here is the output from the SELECT statement:

Output 3.12 *Result Set of Left Outer Qualified Join with an ON Clause*

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

Key Ideas

- A left outer join is requested with the syntax LEFT [OUTER].
- 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.
- This qualified join example 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.

Example of a Left Outer Natural Join

Program

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

Here is the output from the SELECT statement:

Output 3.13 *Result Set of a Left Outer Natural Join*

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

Key Ideas

- 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.
- 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.

Example of Right Outer Qualified Join

Program

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

Here is the output from the SELECT statement:

Output 3.14 *Result Set from Right Outer Qualified Join*

PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
3422	Oat	3421	2	$2,789,654	United States
3234	Rice	3421	2	$2,789,654	United States
1424	Corn	3421	2	$2,789,654	United States
3421	Wheat	3421	2	$2,789,654	United States
3975	Barley	3421	2	$2,789,654	United States
3234	Rice	3234	1	$189,400	United States
1424	Corn	3234	1	$189,400	United States
3421	Wheat	3234	1	$189,400	United States
3422	Oat	3234	1	$189,400	United States
3975	Barley	3234	1	$189,400	United States
.		1424	3	$555,789	Japan
.		3421	4	$781,183	Japan
.		3975	5	$899,453	Argentina

Key Ideas

- A right outer join is requested with the syntax RIGHT [OUTER].

- 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.
- 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.

Example of a Right Outer Natural Join

Program

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

Here is the output from the SELECT statement:

Output 3.15 *Result Set from Right Outer Natural Join*

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

Key Ideas

- 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.
- 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.

Example of a Full Outer Qualified Join

Program

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

Here is the output from the SELECT statement:

Output 3.16 *Result Set from a Full Outer Qualified Join*

PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
3234	Rice	3234	1	$189,400	United States
3234	Rice	1424	3	$555,789	Japan
3234	Rice	3421	4	$781,183	Japan
3234	Rice	3421	2	$2,789,654	United States
3234	Rice	3975	5	$899,453	Argentina
1424	Corn	.	.	.	
3421	Wheat	.	.	.	
3422	Oat	.	.	.	
3975	Barley	.	.	.	

Key Ideas

- A full outer join is requested with the syntax FULL [OUTER]. 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.
- 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.

Example of a Full Outer Natural Join

Program

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

Here is the output from the SELECT statement:

Output 3.17 *Result Set from Full Outer Natural Join*

PRODID	PRODUCT	CUSTID	TOTALS	COUNTRY
3234	Rice	1	$189,400	United States
1424	Corn	3	$555,789	Japan
3421	Wheat	4	$781,183	Japan
3421	Wheat	2	$2,789,654	United States
3975	Barley	5	$899,453	Argentina
3422	Oat	.	.	

Key Ideas

- 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.
- 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.

Chapter 4

FedSQL Expressions and Subqueries

Overview of FedSQL Expressions and Subqueries

FedSQL for CAS supports numeric and row value expressions in the SELECT statement and subqueries.

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
- ARRAY[ ]
- ROW (*row-value-constructor1*, *row-value-constructor2*, *row-value-constructor2...row-value-constructorN*)
- *row-subquery*

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. This example illustrates the use of the ROW keyword with a row value constructor:

```
select * from WorldTemps where ROW (city, country) = ROW ('Madrid', 'Spain')
```

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).

FedSQL for CAS supports *non-correlated subqueries*. A non-correlated subquery calculates a value from a joined table that is independent of the outer query and uses the value somewhere in the outer 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.

The non-correlated subqueries can appear in various places within the SELECT statement. Here are examples:

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

Scalar subqueries can be specified 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.

FedSQL for CAS does not support use of non-correlated subqueries with the IN, ANY, and ALL predicates. For example, the following non-correlated subquery is not supported:

```
select * from table1 where x in (select x from table2);
```

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

Subquery Examples

General Example of a Scalar Subquery in the WHERE Clause

Program

Probably the most common use of a non-correlated subquery is a scalar subquery in a WHERE or HAVING clause to filter rows coming out of the outer query.

```
select something from table1 where table1.x >
   (select avg(something-else) from table2)
```

Key Ideas

- *Something* is a "<sql-expression>" that selects at least one column from table1.
- A WHERE clause is specified to filter the rows that are returned. The WHERE clause specifies a different column from table1.
- The WHERE clause includes an operator (>) between the inner query and the outer query that serves as a filter.
- The subquery selects a single value from at least one column in table2 (using a "<sql-expression>") that is used as input to the operator.

General Example of a Non-Correlated Subquery in the FROM Clause

Program

Subqueries in the FROM clause are used to package and name an intermediate result set for use in the outer query. The outer query can join, aggregate, sort, or otherwise manipulate the intermediate result. A very common case would be to put a join inside the FROM clause subquery, with calculated values in the SELECT list of that join, and use the outer query to group by the calculated values. Here is an example of such a query. The subquery specifies the SUBSTRING function to create the intermediate result set.

```
select A, max(B)
from
  (select substring(table1.x from 1 for 2) ||
        substring(table2.y from 3 for 2) as A,
```

```
            table1.B
        from table1, table2
        where table1.z=table2.z )  T
    group by A
```

Key Ideas

- The outer SELECT statement specifies two variables: A and max(B).
- The subquery does several things:
  - it uses the SUBSTRING function to select a column value from column X in table1 and column Y in table2 and creates an intermediate result set, which is assigned the alias A.
  - it selects column table1.B, which will later have the MAX function applied.
  - it specifies to join table1 and table2 based on values that they have in common in a column that exists in both tables, X. The join is assigned the alias T.
- The outer query specifies to group the results in T by the value in A.

Specific Example of a Non-Correlated Subquery in the FROM Clause

Program

This example queries tables WORLDTEMPS and WORLDCITYCOORDS by specifying a subquery in the FROM clause. This example uses the subquery to annotate each output row with the sum of the average high for the matching nation.

```
select
    C.*, T.AvgHigh as AvgHighCity, AvgHighNation
  from worldcitycoords C,
    worldtemps T,
    ( select Country, avg(AvgHigh) as AvgHighNation from worldtemps
     group by Country ) AHN
   where T.City = C.City and
    T.Country = AHN.Country
  order by C.Country, C.City
```

Here is the output from the SELECT statement:

Output 4.1 *Results of Query on Tables WorldCityCoords and WorldTemps*

CITY	COUNTRY	LATITUDE	LONGITUDE	AVGHIGHCITY	AVGHIGHNATION
Algiers	Algeria	37	3	90	90
Bejing	China	40	116	86	87.5
Hong Kong	China	22	114	89	87.5
Shanghai	China	31	121	.	87.5
Bombay	India	19	73	90	93.5
Calcutta	India	22	88	97	93.5
Amsterdam	Netherlands	52	5	79	79
Lagos	Nigeria	6	3	90	90
Madrid	Spain	40	4	89	89
Zurich	Switzerland	47	8	78	77
Caracas	Venezuela	10	67	83	83

Key Ideas

- The outer query selects all columns from table WorldCityCoords. the AvgHigh column from table WorldTemps (and names it AvgHighCity), and specifies a new column named AvgHighNation.
- The subquery invokes the AVG function on column AvgHigh from WorldTemps to create column AvgHighNation and specifies to group the results by Country. The output from the subquery is assigned the variable AHN.
- The outer query specifies to join tables WorldCityCoords and WorldTemps based on the values of the column City, which they have in common, as well as the Country values that table WorldTemps and output variable AHN have in common.
- The outer query orders the results of the equijoin by City and Country.

Part 2

FedSQL Reference

Chapter 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.

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>" on page 58

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 customer numbers beyond 15.

Example

```
select * from invtry
   where invtry.name
      between 'A' and 'Mzzz';
```

See Also

Expressions:

- "<sql-expression>" on page 58

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>" on page 58

"Overview of FedSQL Expressions and Subqueries" on page 33

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>" on page 58

result-expression

specifies an SQL expression that evaluates to a value.

See "<sql-expression>" on page 58

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 356

```
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 356

```
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 46
- "NULLIF Expression" on page 57
- <search-condition> in "SELECT Statement" on page 321

COALESCE Expression

Returns the first non-null value from a list of columns.

Restriction: CAS tables process null values as a blank string.

Syntax

COALESCE(*expression* [, …*expression*])

Arguments

expression
specifies any valid SQL expression.

See "<sql-expression>" on page 58

"Overview of FedSQL Expressions and Subqueries" on page 33

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 334.

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 42

DISTINCT Predicate

Specifies that only unique rows can appear in the result table.

Syntax

Form 1: *function* **DISTINCT** (*expression*);

Form 2: **SELECT DISTINCT** <select-list> FROM <table-expression>;

Arguments

function
can be any aggregate function.

expression
specifies any valid SQL expression.

See "<sql-expression>" on page 58

"Overview of FedSQL Expressions and Subqueries" on page 33

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 "SELECT Clause" on page 324.

Details

You can use the DISTINCT predicate to see whether two values or two row values are equal to one another. The DISTINCT predicate evaluates to true only if all rows that its subquery returns are distinct.

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 321

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 321

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 321

IN Predicate

Tests set membership.

Syntax

expression [NOT] **IN** (*constant* [, …*constant*])

Arguments

expression

specifies any valid SQL expression.

Restriction	The IN predicate does not support subqueries.
See	"<sql-expression>" on page 58
	"Overview of FedSQL Expressions and Subqueries" on page 33

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 356

```
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>" on page 58

"Overview of FedSQL Expressions and Subqueries" on page 33

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 355

```
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 54
- "IS UNKNOWN Predicate" on page 55
- <search-condition> in the "SELECT Statement" on page 321

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>" on page 58

"Overview of FedSQL Expressions and Subqueries" on page 33

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 355

```
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 52
- <search-condition> in the "SELECT Statement" on page 321

IS NULL Predicate

Tests for a null value.

Syntax

expression **IS** [NOT] **NULL**

Arguments

expression
specifies any valid SQL expression.

See "<sql-expression>" on page 58

"Overview of FedSQL Expressions and Subqueries" on page 33

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 355

```
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 51
- <search-condition> in the "SELECT Statement" on page 321

IS TRUE Predicate

Tests for a true value.

Syntax

(*expression*) **IS** [NOT] **TRUE**

Arguments

expression
specifies any valid SQL expression.

See "<sql-expression>" on page 58

"Overview of FedSQL Expressions and Subqueries" on page 33

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 355

```
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 50
- "IS UNKNOWN Predicate" on page 55

IS UNKNOWN Predicate

Tests for an unknown value.

Syntax

expression **IS** [NOT] **UNKNOWN**

Arguments

expression
specifies any valid SQL expression.

See "<sql-expression>" on page 58

"Overview of FedSQL Expressions and Subqueries" on page 33

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 50
- "IS TRUE Predicate" on page 54
- <search-condition> in the "SELECT Statement" on page 321

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>" on page 58

"Overview of FedSQL Expressions and Subqueries" on page 33

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 353

```
select name, population
   from densities
      where name like 'Al%';
```

See Also

Functions:

- "TRIM Function" on page 304

NULLIF Expression

Returns a null value if the two specified expressions are equal; otherwise, returns the first expression.

Restriction: The CAS file format processes a null value as a DOUBLE value in some situations and as a blank string in other situations. For more information, see "Handling of Nonexistent Data" on page 10.

Syntax

NULLIF(*expression-1, expression-2*)

Arguments

expression

specifies any valid SQL expression.

Data type	All data types are valid.
See	"<sql-expression>" on page 58
	"Overview of FedSQL Expressions and Subqueries" on page 33

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 355

```
missingLong= '.L';
update worldcitycoords
set longitude = nullif(missingLong, '.');
select city
   from worldcitycoords
      where Longitude='.L';
```

See Also

Expressions:

- “CASE Expression” on page 42
- “COALESCE Expression” on page 46

<sql-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 Chapter 7, "FedSQL Functions," on page 171

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 42 or "COALESCE Expression" on page 46.

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.

The following predicates are valid.

- "BETWEEN Predicate" on page 41
- "DISTINCT Predicate" on page 47
- "EXISTS Predicate" on page 48
- "IN Predicate" on page 49
- "IS FALSE Predicate" on page 50.
- "IS MISSING Predicate" on page 51
- "IS NULL Predicate" on page 52
- "IS TRUE Predicate" on page 54
- "IS UNKNOWN Predicate" on page 55
- "LIKE Predicate" on page 55

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 (enclosed 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). In the initial FedSQL release for CAS, subqueries are not supported in the IN predicate.

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

Group	Expressions, Operators, and Predicates	Description
1	CASE expression	See “CASE Expression” on page 42
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 41.
	DISTINCT predicate	See “DISTINCT Predicate” on page 47
	[NOT] EXISTS predicate	See “EXISTS Predicate” on page 48
	[NOT] IN predicate	See “IN Predicate” on page 49
	IS [NOT] TRUE predicate	See “IS TRUE Predicate” on page 54
	IS [NOT] FALSE predicate	See “IS FALSE Predicate” on page 50
	IS [NOT] MISSING predicate	See “IS MISSING Predicate” on page 51
	IS [NOT] NULL predicate	See “IS NULL Predicate” on page 52
	IS [NOT] UNKNOWN predicate	See “IS UNKNOWN Predicate” on page 55
	LIKE predicate	See “LIKE Predicate” on page 55
7	=	equals
	^=, <>	does not equal
	>	is greater than
	<	is less than
	>=	is greater than or equal to

Group	Expressions, Operators, and Predicates	Description
	<=	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 321
- “Overview of FedSQL Expressions and Subqueries” on page 33

Chapter 6

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, Scalable Performance Data (SPD) Engine data set, or SASHDAT file with FedSQL, formatting instructions can be stored with the data set, and automatically applied by the SAS Output Delivery System 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 *SAS Viya Data Management and Utility 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 in order for the BESTw. format to be applied. FedSQL prints blanks if you do not specify an adequate width. To illustrate, consider the following request:

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

The code returns the output **1E4**. Meanwhile, the following request returns a blank value.

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

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 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 write 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 71.
- 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 VARCHAR. 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 VARCHAR.
- 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: "PUT Function" on page 275.

Validation of FedSQL Formats

The PUT function validates the specified format upon use.

FedSQL Format Examples

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

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

Using a User-Defined Format

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 or SPD Engine data set, 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 change the numeric department codes in the DEPT column of the EMPLOYEES table to their corresponding character-based department name.

```
select emp_name, hire_date, put(dept, deptno.) as dept
from employees limit 4;
quit;
```

The content of the source Employees table is shown in Figure 6.1 on page 67. The output of the PUT function is shown in Figure 6.2 on page 68.

Figure 6.1 *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 6.2 *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 *SAS Viya Data Management and Utility Procedures Guide*.

SAS Output Delivery System and FedSQL

The SAS Output Delivery System (ODS), which PROC FEDSQL uses to display results, by default rounds numeric output to appear inside 8 spaces. 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;
```

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:

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 NLS features in SAS 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 Viya National Language Support: 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.
	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.

Category	Language Element	Description
	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 a SAS datetime value to the day of the week, date, and time of the specified locale.
	NLDATMWN*w*.	Converts a SAS datetime value to the day of the week of the specified locale.
	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 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.

Category	Language Element	Description
	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.
	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
Character	$BASE64X*w*. Format (p. 76)	Converts character data into ASCII text by using Base 64 encoding.
	$BINARY*w*. Format (p. 77)	Converts character data to binary representation.
	$CHAR*w*. Format (p. 78)	Writes standard character data.
	$HEX*w*. Format (p. 79)	Converts character data to hexadecimal representation.
	$OCTAL*w*. Format (p. 80)	Converts character data to octal representation.
	$QUOTE*w*. Format (p. 81)	Writes data values that are enclosed in single quotation marks.

Category	Language Elements	Description
	$REVERJ*w*. Format (p. 82)	Writes character data in reverse order and preserves blanks.
	$REVERS*w*. Format (p. 83)	Writes character data in reverse order and left aligns.
	$UPCASE*w*. Format (p. 84)	Converts character data to uppercase.
	$*w*. Format (p. 85)	Writes standard character data.
Date and Time	DATE*w*. Format (p. 93)	Writes SAS date values in the form ddmmmyy, ddmmmyyyy, or dd-mmm-yyyy.
	DATEAMPM*w.d* Format (p. 95)	Writes SAS datetime values in the form ddmmmyy:hh:mm:ss.ss with AM or PM.
	DATETIME*w.d* Format (p. 96)	Writes SAS datetime values in the form ddmmmyy:hh:mm:ss.ss.
	DAY*w*. Format (p. 98)	Writes SAS date values as the day of the month.
	DDMMYY*w*. Format (p. 99)	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.
	DDMMYY*xw*. Format (p. 101)	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.
	DOWNAME*w*. Format (p. 105)	Writes SAS date values as the name of the day of the week.
	DTDATE*w*. Format (p. 106)	Expects a SAS datetime value as input and writes the SAS date values in the form ddmmmyy or ddmmmyyyy.
	DTMONYY*w*. Format (p. 107)	Writes the date part of a SAS datetime value as the month and year in the form mmmyy or mmmyyyy.
	DTWKDATX*w*. Format (p. 108)	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).
	DTYEAR*w*. Format (p. 109)	Writes the date part of a SAS datetime value as the year in the form yy or yyyy.
	DTYYQC*w*. Format (p. 110)	Writes the date part of a SAS datetime value as the year and the quarter, and separates them with a colon (:).
	HHMM*w.d* Format (p. 118)	Writes SAS time values as hours and minutes in the form hh:mm.
	HOUR*w.d* Format (p. 120)	Writes SAS time values as hours and decimal fractions of hours.
	JULIAN*w*. Format (p. 122)	Writes SAS date values as Julian dates in the form yyddd or yyyyddd.

Category	Language Elements	Description
	MMDDYY*w*. Format (p. 123)	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.
	MMDDYY*xw*. Format (p. 124)	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.
	MMSS*w.d* Format (p. 126)	Writes SAS time values as the number of minutes and seconds since midnight.
	MMYY*w*. Format (p. 127)	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.
	MMYY*xw*. Format (p. 128)	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.
	MONNAME*w*. Format (p. 130)	Writes SAS date values as the name of the month.
	MONTH*w*. Format (p. 131)	Writes SAS date values as the month of the year.
	MONYY*w*. Format (p. 132)	Writes SAS date values as the month and the year in the form mmmyy or mmmyyyy.
	NENGO*w*. Format (p. 134)	Writes SAS date values as Japanese dates in the form e.yymmdd.
	QTR*w*. Format (p. 138)	Writes SAS date values as the quarter of the year.
	QTRR*w*. Format (p. 139)	Writes SAS date values as the quarter of the year in roman numerals.
	TIME*w.d* Format (p. 142)	Writes SAS time values as hours, minutes, and seconds in the form hh:mm:ss.ss using the military 24-hour clock.
	TIMEAMPM*w.d* Format (p. 143)	Writes SAS time values as hours, minutes, and seconds in the form hh:mm:ss.ss with AM or PM.
	TOD*w.d* Format (p. 145)	Writes SAS time values and the time portion of SAS datetime values in the form hh:mm:ss.ss.
	WEEKDATE*w*. Format (p. 148)	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).
	WEEKDATX*w*. Format (p. 150)	Writes SAS date values as the day of the week and date in the form day-of-week, dd month-name yy (or yyyy).
	WEEKDAY*w*. Format (p. 151)	Writes SAS date values as the day of the week.
	YEAR*w*. Format (p. 152)	Writes SAS date values as the year.

Category	Language Elements	Description
	YYMM*w*. Format (p. 154)	Writes SAS date values in the form [yy]yyMmm, where M is the separator and the year appears as either 2 or 4 digits.
	YYMM*xw*. Format (p. 155)	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.
	YYMMDD*w*. Format (p. 157)	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.
	YYMMDD*xw*. Format (p. 159)	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 that 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.
	YYMON*w*. Format (p. 160)	Writes SAS date values in the form yymmm or yyyymmm.
	YYQ*w*. Format (p. 161)	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.
	YYQ*xw*. Format (p. 163)	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.
	YYQR*w*. Format (p. 165)	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.
	YYQR*xw*. Format (p. 166)	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.
	YYQZ*w*. Format (p. 168)	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	BEST*w*. Format (p. 86)	SAS chooses the best notation.
	BESTD*w.p* Format (p. 87)	Prints numeric values, lining up decimal places for values of similar magnitude, and prints integers without decimals.
	BINARY*w*. Format (p. 88)	Converts numeric values to binary representation.
	COMMA*w.d* Format (p. 89)	Writes numeric values with a comma that separates every three digits and a period that separates the decimal fraction.

Category	Language Elements	Description
	COMMAX*w.d* Format (p. 90)	Writes numeric values with a period that separates every three digits and a comma that separates the decimal fraction.
	D*w.p* Format (p. 92)	Prints variables, possibly with a great range of values, lining up decimal places for values of similar magnitude.
	DOLLAR*w.d* Format (p. 102)	Writes numeric values with a leading dollar sign, a comma that separates every three digits, and a period that separates the decimal fraction.
	DOLLARX*w.d* Format (p. 104)	Writes numeric values with a leading dollar sign, a period that separates every three digits, and a comma that separates the decimal fraction.
	E*w*. Format (p. 111)	Writes numeric values in scientific notation.
	EURO*w.d* Format (p. 112)	Writes numeric values with a leading euro symbol (€), a comma that separates every three digits, and a period that separates the decimal fraction.
	EUROX*w.d* Format (p. 114)	Writes numeric values with a leading euro symbol (€), a period that separates every three digits, and a comma that separates the decimal fraction.
	FLOAT*w.d* Format (p. 115)	Generates a native single-precision, floating-point value by multiplying a number by 10 raised to the dth power.
	FRACT*w*. Format (p. 116)	Converts numeric values to fractions.
	HEX*w*. Format (p. 117)	Converts real binary (floating-point) values to hexadecimal representation.
	IEEE*w.d* Format (p. 121)	Generates an IEEE floating-point value by multiplying a number by 10 raised to the dth power.
	NEGPAREN*w.d* Format (p. 133)	Writes negative numeric values in parentheses.
	OCTAL*w*. Format (p. 135)	Converts numeric values to octal representation.
	PERCENT*w.d* Format (p. 136)	Writes numeric values as percentages.
	PERCENTN*w.d* Format (p. 137)	Produces percentages, using a minus sign for negative values.
	ROMAN*w*. Format (p. 140)	Writes numeric values as roman numerals.
	SIZEK*w.d* Format (p. 141)	Writes a numeric value in the form nK for kilobytes.
	VAXRB*w.d* Format (p. 146)	Writes real binary (floating-point) data in VMS format.
	w.d Format (p. 147)	Writes standard numeric data one digit per byte.

Category	Language Elements	Description
	YEN*w.d* Format (p. 153)	Writes numeric values with yen signs, commas, and decimal points.
	Z*w.d* Format (p. 169)	Writes standard numeric data with leading 0s.

Dictionary

$BASE64X*w*. Format

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

Category: Character

Alignment: Left

Syntax

$BASE64X*w*.

Arguments

w

specifies the width of the output field.

Default 1

Range 1–32767

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

Statements	Results
`select put ("FCA01A7993BC", $base64x64.);`	`RkNBMDFBNzk5M0JD`
`select put ("MyPassword", $base64x64.);`	`TXlQYXNzd29yZA==`
`select put ("www.mydomain.com/myhiddenURL", $base64x64.);`	`d3d3Lm15ZG9tYWluLmNvbi9teWhpZGRlblVSTA==`

$BINARY*w*. Format

Converts character data to binary representation.

Category: Character

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

Statements	Results	
	ASCII	EBCDIC
	`----+----1----+----2`	`----+----1----+----2`
`select put('AB', $binary16.);`	`0100000101000010`	`1100000111000010`

See Also

Formats:

- "BINARYw. Format" on page 88
- "$HEXw. Format" on page 79

$CHAR*w*. Format

Writes standard character data.

Category: Character

Alignment: Left

Alias: $*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*. and $*w*. formats are identical, and they do not trim leading blanks.

Example

Statements	Results
	`----+----1`
`select put ('XYZ',$char.);`	`XYZ`

See Also

Formats:

- "$w. Format" on page 85

$HEX*w*. Format

Converts character data to hexadecimal representation.

Category: Character

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

Statements	Results
`select put ('AB',$hex5.);`	`----+----1----+----2` `4142`

See Also

Formats:

- "$BINARYw. Format" on page 77
- "HEXw. Format" on page 117

$OCTAL*w*. Format

Converts character data to octal representation.

Category: Character

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

Statements	Results
	`----+----1----+----2`
`select put ('art', $octal15.);`	`141162164040040`
`select put ('rice', $octal15.);`	`162151143145040`
`select put ('bank', $octal15.);`	`162151143145040`

See Also

Formats:

- "OCTALw. Format" on page 135

$QUOTE*w*. Format

Writes data values that are enclosed in single quotation marks.

Category: Character

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.

- If your data value is not enclosed in quotation marks, SAS encloses the output in double quotation marks.
- If 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.
- If 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.
- If 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.

- If 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.
- If the length of the target field is not large enough to contain the string and its quotation marks, SAS returns all blanks.

Example

Statements	Results
	`----+----1----+----2`
`select put('SAS',$quote.);`	`"SAS"`
`select put('SAS''s',$quote.);`	`"SAS's"`
`select put('ad''verb',$quote16.);`	`"ad'verb"`
`select put('"ad" "verb"',$quote16.);`	`"""ad""""verb"""`
`select put('"ad"'"verb"',$quote20);`	`"""ad"""'""verb"""`

$REVERJ*w*. Format

Writes character data in reverse order and preserves blanks.

Category: Character

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

Statements*	Results*
	`----+----1`
`select put('ABCD###',$reverj7.);`	`###DCBA`
`select put('###ABCD',$reverj7.);`	`DCBA###`

* The character # represents a blank space.

See Also

Formats:

- "$REVERSw. Format" on page 83

$REVERS*w*. Format

Writes character data in reverse order and left aligns.

Category: Character

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

Statements*	Results
	`----+----1`
`select put('ABCD###',$revers7);`	DCBA
`select put('###ABCD',$revers7.);`	DCBA

* The character # represents a blank space.

See Also

Formats:

- "$REVERJw. Format" on page 82

$UPCASE*w*. Format

Converts character data to uppercase.

Category: Character

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

Statements	Results
	`----+----1`
`select put('coxe-ryan',upcase9.);`	COXE-RYAN

$w. Format

Writes standard character data.

Category: Character

Alignment: Left

Alias: $F*w*.

Syntax

$*w*.

Arguments

w

specifies the width of the output field.

Default 1 if the length of the variable is undefined; otherwise, the length of the variable.

Range 1–32767

Comparisons

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

Example

Statements*	Results*
	`----+----1----+----2`
`select put('#Cary',$5.);`	`#Cary`
`select put('#Cary',$f5.);`	`#Cary`
`select put(Carolina,$5.);`	`Carol`

* The character # represents a blank space.

See Also

Formats:

- "$CHARw. Format" on page 78
- "w.d Format" on page 147

BEST*w*. Format

SAS chooses the best notation.

Category: Numeric

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

Statements	Results
	`----+----1----+----2`
`select put(1257000,best6.);`	`1.26E6`

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

See Also

Formats:

- "BESTDw.p Format" on page 87
- "Dw.p Format" on page 92
- "w.d Format" on page 147

BESTD*w.p* Format

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

Category: Numeric

Alignment: Right

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

Statements	Results
	----+----1----+
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 86
- “Dw.p Format” on page 92
- “w.d Format” on page 147

BINARY*w*. Format

Converts numeric values to binary representation.

Category: Numeric

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

Statements	Results
	`----+----1`
`select put (123.45, binary8.);`	`01111011`
`select put (123, binary8.);`	`01111011`
`select put (-123, binary8.);`	`10000101`

See Also

Formats:

- "$BINARYw. Format" on page 77
- "HEXw. Format" on page 117

COMMA*w.d* Format

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

Category: Numeric

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

Statements	Results
	`----+----1----+----2`
`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 90
- "DOLLARw.d Format" on page 102

COMMAX*w.d* Format

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

Category: Numeric

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

Statements	Results
	`----+----1----+----2`
`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 89
- “DOLLARXw.d Format” on page 104

D*w.p* Format

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

Category: Numeric

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

Statements	Results
	----+----1----+----2
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 86
- "BESTDw.p Format" on page 87
- "w.d Format" on page 147

DATE*w*. Format

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

Category: Date and Time

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
	`----+----1----+`
`select put (19431,date5.);`	`14MAR`
`select put (19431,date6.);`	` 14MAR`
`select put (19431,date7.);`	`14MAR13`
`select put (19431,date8.);`	` 14MAR13`
`select put (19431,date9.);`	`14MAR2013`

See Also

Formats:

- "DATETIMEw.d Format" on page 96
- "DAYw. Format" on page 98
- "DDMMYYw. Format" on page 99
- "DTDATEw. Format" on page 106
- "JULIANw. Format" on page 122
- "MMDDYYw. Format" on page 123
- "MONTHw. Format" on page 131
- "NENGOw. Format" on page 134
- "WEEKDATEw. Format" on page 148
- "YEARw. Format" on page 152

Functions:

- "CURRENT_DATE Function" on page 203
- "DATEPART Function" on page 211
- "DAY Function" on page 213

DATEAMPM*w.d* Format

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

Category: Date and Time

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
	`----+----1----+----2----+`
`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 96
- "TIMEAMPMw.d Format" on page 143

DATETIME*w.d* Format

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

Category: Date and Time

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
	`----+----1----+----2`

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 95
- "DDMMYYxw. Format" on page 101
- "DDMMYYw. Format" on page 99

DAY*w*. Format

Writes SAS date values as the day of the month.

Category: Date and Time

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
	`----+----1`
`select put(19523,day2.);`	`14`

See Also

Formats:

- "DATEw. Format" on page 93

Functions:

- "DAY Function" on page 213

DDMMYY*w*. 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.

Category: Date and Time

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 that ranges from 2 to 5, the date appears with as many digits 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
	`----+----1----+`
`select put(19704,ddmmyy.);`	`12/12/13`
`select put(19704,ddmmyy5.);`	`12/13`
`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 93
- “DDMMYYw. Format” on page 99
- “MMDDYYw. Format” on page 123
- “YYMMDDw. Format” on page 157

Functions:

- “DAY Function” on page 213
- “MONTH Function” on page 256
- “YEAR Function” on page 315

DDMMYY*xw*. 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.

Category: Date and Time

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 that ranges from 2 to 5, the date appears with as many digits 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
	`----+----1----+`
`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 93
- "DDMMYYw. Format" on page 99
- "MMDDYYxw. Format" on page 124
- "YYMMDDxw. Format" on page 159

Functions:

- "DAY Function" on page 213
- "MONTH Function" on page 256
- "YEAR Function" on page 315

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.

Category: Numeric

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

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

See Also

Formats:

- "COMMAw.d Format" on page 89
- "DOLLARXw.d Format" on page 104
- "EUROw.d Format" on page 112

• "YENw.d Format" on page 153

DOLLARX*w.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.

Category: Numeric

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

Statements	Results
	----+----1----+
select put(1254.71,dollarx10.2);	$1.254,71

See Also

Formats:

- "COMMAXw.d Format" on page 90
- "DOLLARw.d Format" on page 102

DOWNAME*w*. Format

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

Category: Date and Time

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
	----+----1
select put(19431,downame.);	Monday

See Also

Formats:

- "DATEw. Format" on page 93
- "DTWKDATXw. Format" on page 108
- "WEEKDATEw. Format" on page 148
- "WEEKDATXw. Format" on page 150
- "WEEKDAYw. Format" on page 151

DTDATE*w*. Format

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

Category: Date and Time

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 1510285759, which is the SAS datetime value that corresponds to 3:49:19 AM on November 10, 2013, and prints both a two-digit and a four-digit year for the DTDATE*w*. format.

Statements	Results
	`----+----1`
`select put(1510285759,dtdate.);`	`10NOV13`
`select put(1510285759,dtdate9.);`	`10NOV2013`

See Also

Formats:

- "DATEw. Format" on page 93

DTMONYY*w*. Format

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

Category: Date and Time

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 1678898894, which is the SAS datetime value that corresponds to March 14, 2013, at 04:48:14 PM.

Statements	Results
	`----+----1`
`select put(1678898894,dtmonyy.);`	`MAR13`
`select put(1678898894,dtmonyy5.);`	`MAR13`
`select put(1678898894,dtmonyy6.);`	`MAR13`
`select put(1678898894,dtmonyy7.);`	`MAR2013`

See Also

Formats:

- "DATETIMEw.d Format" on page 96
- "MONYYw. Format" on page 132

DTWKDATX*w*. 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*).

Category: Date and Time

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 1678898894, which is the SAS datetime value that corresponds to March 14, 2013, at 04:48:14 PM.

Statements	Results
	`----+----1----+----2----+----3`
`select put(1678898894,dtwkdatx.);`	`Saturday, 14 March 2013`
`select put(1678898894,dtwkdatx3.);`	`Sat`
`select put(1678898894,dtwkdatx8.);`	`Sat`
`select put(1678898894,dtwkdatx25.);`	`Saturday, 14 Mar 2013`

See Also

Formats:

- "DATEw. Format" on page 93
- "WEEKDATEw. Format" on page 148
- "WEEKDATXw. Format" on page 150

DTYEAR*w*. Format

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

Category: Date and Time

Alignment: Right

Syntax

DTYEAR*w*.

Arguments

w

specifies the width of the output field.

Default 4

Range 2–4

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 1678898894, which is the SAS datetime value that corresponds to March 14, 2013, at 04:48:14 AM.

Statements	Results
	`----+----1`
`select put(1678898894,dtyear.);`	`2013`
`select put(1678898894,dtyear2.);`	`13`
`select put(1678898894,dtyear3.);`	`13`
`select put(1678898894,dtyear4.);`	`2013`

See Also

Formats:

- "DATEw. Format" on page 93
- "DATETIMEw.d Format" on page 96
- "YEARw. Format" on page 152

DTYYQC*w*. Format

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

Category: Date and Time

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 1678898894, which is the SAS datetime value that corresponds to March 14, 2013, at 04:48:52 PM.

Statements	Results
	`----+----1`
`select put(1678898894,dtyyqc.);`	`13:1`
`select put(1678898894,dtyyqc4.);`	`13:1`
`select put(1678898894,dtyyqc5.);`	`13:1`
`select put(1678898894,dtyyqc6.);`	`2013:1`

See Also

Formats:

- "DATEw. Format" on page 93
- "DATETIMEw.d Format" on page 96

E*w*. Format

Writes numeric values in scientific notation.

Category: Numeric

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

Statements	Results
	`----+----1----+`
`select put(1257,e10.);`	`1.257E+03`
`select put(-1257,e10.);`	`-1.257E+03`

EURO*w.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.

Category: Numeric

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
	`----+----1----+----2----+----3`
`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 102
- “EUROXw.d Format” on page 114
- “YENw.d Format” on page 153

EUROX*w.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.

Category: Numeric

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
	----+----1----+----2----+----3
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 112

FLOAT*w.d* Format

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

Category: Numeric

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

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

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

See Also

Formats:

- "IEEEw.d Format" on page 121

FRACT*w*. Format

Converts numeric values to fractions.

Category: Numeric

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, it writes 1/2 instead of 50/100.

Example

Statements	Results
	`----+----1`

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

HEX*w*. Format

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

Category: Numeric

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

Statements	Results
	----+----1----+----2
select put(35.4, hex8.);	00000023
select put(88, hex8.);	00000058
select put(2.33, hex8.);	00000002
select put(-150, hex8.);	FFFFFF6A

See Also

Formats:

- "BINARYw. Format" on page 88
- "$HEXw. Format" on page 79

HHMM*w.d* Format

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

Category: Date and Time

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
	`----+----1`
`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 120
- "MMSSw.d Format" on page 126
- "TIMEw.d Format" on page 142
- "TODw.d Format" on page 145

Functions:

- "HOUR Function" on page 225
- "MINUTE Function" on page 254
- "SECOND Function" on page 287

HOUR*w.d* Format

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

Category: Date and Time

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
	`----+----1`
`select put(41400,hour4.1);`	`11.5`

See Also

Formats:

- “HHMMw.d Format” on page 118
- “MMSSw.d Format” on page 126
- “TIMEw.d Format” on page 142

• "TODw.d Format" on page 145

IEEE*w.d* Format

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

Category: Numeric

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

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

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

See Also

Formats:

- "FLOATw.d Format" on page 115

JULIAN*w*. Format

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

Category: Date and Time

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
	`----+----1`
`select put(19431,julian5.);`	`13073`
`select put(19431,julian7.);`	`2013073`

See Also

Formats:

- "DATEw. Format" on page 93

MMDDYY*w*. 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.

Category: Date and Time

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 that ranges from 2 to 5, the date appears with as many digits 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
	`----+----1----+`
`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 93
- “DDMMYYw. Format” on page 99
- “MMDDYYxw. Format” on page 124
- “YYMMDDw. Format” on page 157

MMDDYY*xw*. 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.

Category: Date and Time

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 that ranges from 2 to 5, the date appears with as many digits 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
	`----+----1----+`
`select put(19431,mmddyyc5.);`	`03:14`
`select put(19431,mmddyyd8.)`	`03-14-13`
`select put(19431,mmddyyn8.);`	`03132013`
`select put(19431,mmddyyp10.)`	`03.14.2013`

See Also

Formats:

- "DATEw. Format" on page 93
- "DDMMYYxw. Format" on page 101
- "MMDDYYw. Format" on page 123
- "YYMMDDxw. Format" on page 159

MMSS*w.d* Format

Writes SAS time values as the number of minutes and seconds since midnight.

Category: Date and Time

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
	`----+----1`
`select put(4530,mmss.);`	`75:30`

See Also

Formats:

- "HHMMw.d Format" on page 118
- "TIMEw.d Format" on page 142

MMYY*w*. 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.

Category: Date and Time

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
	`----+----1----+`
`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 93
- "MMYYxw. Format" on page 128
- "YYMMw. Format" on page 154

MMYY*xw*. 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.

Category: Date and Time

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
	`----+----1----+`
`select put(19560,mmyyc5.);`	`07:13`

Statements	Results
`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 93
- "MMYYw. Format" on page 127
- "YYMMw. Format" on page 154

MONNAME*w*. Format

Writes SAS date values as the name of the month.

Category: Date and Time

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
	`----+----1`
`select put(19431,monname1.);`	`M`
`select put(19431,monname3.);`	`Mar`
`select put(19431,monname5.);`	`March`

See Also

Formats:

- "MONTHw. Format" on page 131

MONTH*w*. Format

Writes SAS date values as the month of the year.

Category: Date and Time

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
	`----+----1`

Statements	Results
`select put(19431,month.);`	3

See Also

Formats:

- "MONNAMEw. Format" on page 130

MONYY*w*. Format

Writes SAS date values as the month and the year in the form *mmmyy* or *mmmyyyy*.

Category: Date and Time

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
	`----+----1`
`select put(19704,monyy5.);`	`DEC13`
`select put(19704,monyy7.);`	`DEC2013`

See Also

Formats:

- "DATEw. Format" on page 93
- "DTMONYYw. Format" on page 107
- "DDMMYYw. Format" on page 99
- "MMDDYYw. Format" on page 123
- "YYMMDDw. Format" on page 157

Functions:

- "MONTH Function" on page 256
- "YEAR Function" on page 315

NEGPAREN*w.d* Format

Writes negative numeric values in parentheses.

Category: Numeric

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 nonnegative, 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

Statements	Results
	`----+----1----+`
`select put(100,negparen6.);`	`  100`
`select put(1000,negparen6.);`	`1,000`
`select put(-200,negparen6.);`	`(200)`
`select put(-2000,negparen8.);`	`(2,000)`

NENGO*w*. Format

Writes SAS date values as Japanese dates in the form *e.yymmdd*.

Category: Date and Time

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 emperor (Meiji, Taisho, Showa, or Heisei).

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
	`----+----1`
`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 93

OCTAL*w*. Format

Converts numeric values to octal representation.

Category: Numeric

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

Statements	Results
	`----+----1`
`select put (3592, octal16.);`	`007010`

See Also

Formats:

- "$OCTALw. Format" on page 80

PERCENT*w.d* Format

Writes numeric values as percentages.

Category: Numeric

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, and formats them the same as the BEST*w.d* format. However, it also adds a percent sign (%) to the end of the formatted value and 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

Statements	Results
	`----+----1----+----2`
`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 137

PERCENTN*w.d* Format

Produces percentages, using a minus sign for negative values.

Category: Numeric

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

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 136

QTR*w*. Format

Writes SAS date values as the quarter of the year.

Category: Date and Time

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 19431, which is the SAS date value that corresponds to March 14, 2013.

Statements	Results
	`----+----1`
`select put(19431,qtr.);`	`1`

See Also

Formats:

- "DATEw. Format" on page 93
- "QTRRw. Format" on page 139
- "YYQw. Format" on page 161
- "YYQxw. Format" on page 163
- "YYQZw. Format" on page 168

QTRR*w*. Format

Writes SAS date values as the quarter of the year in roman numerals.

Category: Date and Time

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
	`----+----1`
`select put(19624,qtrr.);`	`III`

See Also

Formats:

- "QTRw. Format" on page 138
- "YYQRw. Format" on page 165
- "YYQRxw. Format" on page 166

ROMAN*w*. Format

Writes numeric values as roman numerals.

Category: Numeric

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

Statements	Results
	`----+----1`
`select put(2006,roman.);`	`MMVI`

See Also

Formats:

- "DATEw. Format" on page 93
- "JULIANw. Format" on page 122

SIZEK*w.d* Format

Writes a numeric value in the form *n*K for kilobytes.

Category: Numeric

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

Statements	Results
	`----+----1`
`select put (1024, sizek.);`	`1K`
`select put (200943, sizek.);`	`197K`

TIME*w.d* Format

Writes SAS time values as hours, minutes, and seconds in the form *hh:mm:ss.ss* using the military 24-hour clock.

Category: Date and Time

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
	`----+----1`
`select put(59083,time.);`	`16:24:43`

See Also

Formats:

- "HHMMw.d Format" on page 118
- "HOURw.d Format" on page 120
- "MMSSw.d Format" on page 126
- "TODw.d Format" on page 145

Functions:

- "HOUR Function" on page 225
- "MINUTE Function" on page 254
- "SECOND Function" on page 287

TIMEAMPM*w.d* Format

Writes SAS time values as hours, minutes, and seconds in the form *hh:mm:ss.ss* with AM or PM.

Category: Date and Time

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
	`----+----1----+`

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 95
- "TIMEw.d Format" on page 142

TOD*w.d* Format

Writes SAS time values and the time portion of SAS datetime values in the form *hh:mm:ss.ss*.

Category: Date and Time

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

Statements	Results
`select put(1472049623,tod9.);`	`14:40:23`

See Also

Formats:

- "TIMEw.d Format" on page 142
- "TIMEAMPMw.d Format" on page 143

VAXRB*w.d* Format

Writes real binary (floating-point) data in VMS format.

Category: Numeric

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

Statements	Results*
	`----+----1`
`select put(1,vaxrb8.);`	`8040000000000000`

* The result is the hexadecimal representation for the integer.

w.d Format

Writes standard numeric data one digit per byte.

Category: Numeric

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

Statements	Results
	`----+----1----+`
`select put (23.45, 6.3);`	`23.450`

See Also

Formats:

- "BESTw. Format" on page 86
- "BESTDw.p Format" on page 87
- "Dw.p Format" on page 92
- "$w. Format" on page 85
- "Zw.d Format" on page 169

WEEKDATE*w*. 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*).

Category: Date and Time

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
	`----+----1----+----2`
`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 108
- “DATEw. Format” on page 93
- “DDMMYYw. Format” on page 99
- “MMDDYYw. Format” on page 123
- “TODw.d Format” on page 145
- “WEEKDATXw. Format” on page 150
- “YYMMDDw. Format” on page 157

WEEKDATX*w*. 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*).

Category: Date and Time

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
	`----+----1----+----2----+----3`
`select put(19405,weekdatx.);`	`Saturday, 16 February 2013`

See Also

Formats:

- "DTWKDATXw. Format" on page 108
- "DATEw. Format" on page 93
- "DDMMYYw. Format" on page 99
- "MMDDYYw. Format" on page 123
- "TODw.d Format" on page 145
- "WEEKDATEw. Format" on page 148
- "YYMMDDw. Format" on page 157

WEEKDAY*w*. Format

Writes SAS date values as the day of the week.

Category: Date and Time

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
	`----+----1`
`select put(19405,weekday.);`	`7`

See Also

Formats:

- "DOWNAMEw. Format" on page 105

YEAR*w*. Format

Writes SAS date values as the year.

Category: Date and Time

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 are printed. Otherwise, the year values are printed 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
	`----+----1`
`select put(19537,year2.);`	`13`
`select put(19537,year4.);`	`2013`

See Also

Formats:

- "DTYEARw. Format" on page 109

YEN*w.d* Format

Writes numeric values with yen signs, commas, and decimal points.

Category: Numeric

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

Statements	Results
	`----+----1`
`select put(1254.71,yen10.2);`	`¥1,254.71`

See Also

Formats:

- "DOLLARw.d Format" on page 102
- "EUROw.d Format" on page 112

YYMM*w*. 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.

Category: Date and Time

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
	`----+----1----+`
`select put(19656,yymm5.);`	`13M10`
`select put(19656,yymm6.);`	`13M10`
`select put(19656,yymm.);`	`2013M10`
`select put(19656,yymm7.);`	`2013M10`
`select put(19656,yymm10.);`	`2013M10`

See Also

Formats:

- "DATEw. Format" on page 93
- "MMYYw. Format" on page 127
- "YYMMxw. Format" on page 155

YYMM*xw*. 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.

Category: Date and Time

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
	`----+----1----+`
`select put(19537,yymmc5.);`	`13:06`
`select put(19537,yymmd.);`	`2013-06`
`select put(19537,yymmn4.);`	`1306`
`select put(19537,yymmp8.);`	`2013.06`

Statements	Results
`select put(19537,yymms10.);`	`2013/06`

See Also

Formats:

- "DATEw. Format" on page 93
- "MMYYxw. Format" on page 128
- "YYMMw. Format" on page 154

YYMMDD*w*. 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.

Category: Date and Time

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 that ranges from 2 to 5, the date appears with as many digits 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
	`----+----1----+`
`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 93
- “DDMMYYw. Format” on page 99
- “MMDDYYw. Format” on page 123
- “YYMMDDxw. Format” on page 159

Functions:

- “DAY Function” on page 213
- “MONTH Function” on page 256
- “YEAR Function” on page 315

YYMMDD*xw*. 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 that 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.

Category: Date and Time

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 that ranges from 2 to 5, the date appears with as many digits of the year 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 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
	`----+----1----+`
`select put(19704,yymmddc5.);`	`13:12`
`select put(19704,yymmddd8.);`	`13-12-12`
`select put(19704,yymmddn8.);`	`20131212`
`select put(19704,yymmddp10.);`	`2013.12.12`

See Also

Formats:

- “DATEw. Format” on page 93
- “DDMMYYxw. Format” on page 101
- “MMDDYYxw. Format” on page 124
- “YYMMDDw. Format” on page 157

Functions:

- “DAY Function” on page 213
- “MONTH Function” on page 256
- “YEAR Function” on page 315

YYMON*w*. Format

Writes SAS date values in the form *yymmm* or *yyyymmm*.

Category: Date and Time

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
	`----+----1`
`select put(19537,yymon6.);`	`13JUN`
`select put(19537,yymon7.);`	`2013JUN`

See Also

Formats:

- "DATEw. Format" on page 93
- "MMYYw. Format" on page 127

YYQ*w*. 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.

Category: Date and Time

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
	`----+----1----+`
`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 93
- “YYQxw. Format” on page 163

- "YYQRw. Format" on page 165
- "YYQRxw. Format" on page 166
- "YYQZw. Format" on page 168

YYQ*xw*. 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.

Category: Date and Time

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
	`----+----1----+`
`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 93
- "YYQw. Format" on page 161
- "YYQRw. Format" on page 165
- "YYQRxw. Format" on page 166
- "YYQZw. Format" on page 168

YYQR*w*. 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.

Category: Date and Time

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
	`----+----1----+`
`select put(19537,yyqr6.);`	`13QII`
`select put(19537,yyqr7.);`	`2013QII`
`select put(19537,yyqr.);`	`2013QII`

Statements	Results
`select put(19537,yyqr8.);`	`2013QII`
`select put(19537,yyqr10.);`	`2013QII`

See Also

Formats:

- "YYQw. Format" on page 161
- "YYQRxw. Format" on page 166

YYQR*xw*. 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.

Category: Date and Time

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
	`----+----1----+`
`select put(19721,yyqrc6.);`	`13:IV`
`select put(19721,yyqrd.);`	`2013-IV`
`select put(197214,yyqrn5.);`	`13IV`
`select put(19721,yyqrp8.);`	`2013.IV`
`select put(19721,yyqrs10.);`	`2013/IV`

See Also

Formats:

- "YYQxw. Format" on page 163
- "YYQRw. Format" on page 165

YYQZ*w*. 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.

Category: Date and Time

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
	`----+----1----+`
`select put (19537,yyqz6.);`	`201302`
`select put (19537,yyqz4.);`	`1302`

See Also

Formats:

- "DATEw. Format" on page 93
- "QTRw. Format" on page 138
- "YYQw. Format" on page 161

Zw.d Format

Writes standard numeric data with leading 0s.

Category: Numeric

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*, Z*w*.*d* writes the value without a decimal point.

Details

The Z*w*.*d* format writes standard numeric values one digit per byte and fills in 0s to the left of the data value.

The Z*w*.*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 Z*w*.*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 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

Statement	Result
	`----+----1`
`select put(1350,z8.);`	`00001350`

See Also

Formats:

- "w.d Format" on page 147

Chapter 7

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*.

Note: For the FedSQL functions that are like-named in Base SAS, these functions can take DOUBLE and character arguments and return values, but all character return values are VARCHAR.

General Function Syntax

Aggregate Function Syntax

Here is the syntax for an aggregate function:

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 aggregate functions. For more information, see "Calling Base SAS Functions Instead of FedSQL Aggregate Functions" on page 178.

Non-Aggregate Function Syntax

Here is the syntax of a function:

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 floor(density) from densities;
```
```
select abs((3 * 50) / 5);
```

Specifying Function Values

FedSQL Date, Time, and Datetime Constants

FedSQL supports industry standard conventions for dates, times, and datetimes using the DATE, TIME, and TIMESTAMP data types. Although CAS tables do not support these data types, FedSQL functions require that you specify date and time input values using these conventions. You write FedSQL date, time, or timestamp constants using the following syntax:

```
DATE'yyyy-mm-dd'
TIME 'hh:mm: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
- *ss* is a two-digit second, 00–61
- *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. When months or dates are single values, the 0 (zero) is not required.

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 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.

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

```
date'2008-01-31'
date'2000-1-1'
time'20:44:59'
timestamp'2007-02-02 07:00:00.7569'
```

Other FedSQL Constants and Character Values

When used in a function, FedSQL constants and character strings must be specified within quotation marks.

The INTCK and INTNX functions are examples of functions that take FedSQL constants. In the following example, the INTCK function specifies the constant QTR and two date values. The INTNX function specifies the constants YEAR and SAME and a date value.

```
select intck('qtr',date'1995-01-10',date'1995-07-01');
select put(intnx('year', date'2011-03-15', 5, 'same'), date9.);
```

The COMPRESS and SCAN functions are examples of functions that take character strings:

```
select compress('abc','a');
select scan('This is a string',2);
```

Specifying Constants and Strings in the FedSQL.execDirect Action

The FedSQL.execDirect action accepts FedSQL statements and functions in a quoted string in the Query parameter. When you use the action, the quotation marks used to submit function values must be different from the quotation marks surrounding the input string. Double single quotation marks are recommended for function values. Here are examples of how FedSQL constant values should be specified in the FedSQL.execDirect action:

```
proc cas;
  fedsql.execdirect query='select intck(''qtr'', date''2013-01-10'',
 date''2013-07-01'')';
quit;

proc cas;
  fedsql.execdirect query='select put(intnx(''year'', date''2011-03-15'',
 5, ''same''), date9.)';
quit;


proc cas;
  fedsql.execdirect query='select compress(''abc'',''a'')';
run;

proc cas;
 fedsql.execdirect query='select scan(''This is a string'',2)';
run;
```

Understanding Function Output

FedSQL Date and Time functions return SAS date and time values. A SAS date value is the number of days from January 1, 1960, to a specified date. A SAS time value is the number of seconds from January 1, 1960, to a specified date. The output of these functions is meaningless unless you apply a SAS format to the value with the PUT function. The following example shows how to format the output of the TODAY() function so that the result has meaning:

```
proc cas;
  fedsql.execdirect query='select put(today(),date.)';
```

```
run;
```

For more information, see Chapter 6, “FedSQL Formats,” on page 63.

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.

Here are the FedSQL aggregate functions:

- “AVG Function” on page 190.
- “COUNT Function” on page 201
- “CSS Function” on page 202
- “KURTOSIS Function” on page 243
- “MAX Function” on page 248
- “MIN Function” on page 253
- “NMISS Function” on page 262
- “PROBT Function” on page 273
- “RANGE Function” on page 277
- “SKEWNESS Function” on page 290
- “STD Function” on page 292
- “STDDEV Function” on page 293
- “STDERR Function” on page 294
- “STUDENTS_T Function” on page 295
- “SUM Function” on page 298
- “USS Function” on page 307
- “VARIANCE Function” on page 309

Using the table “WorldTemps” on page 356, the following aggregate function examples operate on the AvgLow table column:

```
/* 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.

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.

Quantile
: returns a quantile from specific distributions.

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.

FEDSQL Functions by Category

Category	Language Elements	Description
Aggregate	AVG Function (p. 190)	Returns the average of all values in a column.
	COUNT Function (p. 201)	Returns the number of rows retrieved by a SELECT statement for a specified table.
	CSS Function (p. 202)	Returns the corrected sum of squares of all values in an expression.
	KURTOSIS Function (p. 243)	Returns the kurtosis of all values in an expression.
	MAX Function (p. 248)	Returns the maximum value in a column.
	MIN Function (p. 253)	Returns the minimum value in an expression.
	NMISS Function (p. 262)	Returns the number of null values or SAS missing values in an expression.
	PROBT Function (p. 273)	Returns the probability from a t distribution of the values in an expression.
	RANGE Function (p. 277)	Returns the range between values in an expression.
	SKEWNESS Function (p. 290)	Returns the skewness of all values in an expression.
	STD Function (p. 292)	Returns the standard deviation.
	STDDEV Function (p. 293)	Returns the statistical standard deviation of all values in an expression.
	STDERR Function (p. 294)	Returns the statistical standard error of all values in an expression.
	STUDENTS_T Function (p. 295)	Returns the Student's t distribution of the values in an expression.
	SUM Function (p. 298)	Returns the sum of all the values in an expression.
	USS Function (p. 307)	Returns the uncorrected sum of squares of all the values in an expression.
	VARIANCE Function (p. 309)	Returns the measure of the dispersion of all values in an expression.
Arithmetic	ERF Function (p. 216)	Returns the value of the (normal) error function.
	ERFC Function (p. 217)	Returns the value of the complementary (normal) error function.

Category	Language Elements	Description
Character	BYTE Function (p. 193)	Returns one character in the ASCII or the EBCDIC collating sequence.
	COMPRESS Function (p. 198)	Returns a character string with specified characters removed from the original string.
	CURRENT_LOCALE Function (p. 204)	Returns the five-character name of the current locale.
	INDEX Function (p. 226)	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. 227)	Searches a character expression for specified characters and returns the position of the first occurrence of any of the characters.
	LOWCASE Function (p. 247)	Converts all letters in a character expression to lowercase.
	RANK Function (p. 278)	Returns the position of a character in the ASCII or EBCDIC collating sequence.
	REPEAT Function (p. 279)	Repeats a character expression.
	REVERSE Function (p. 280)	Reverses a character expression.
	SCAN Function (p. 285)	Returns the nth word from a character expression.
	SUBSTRING Function (p. 297)	Extracts a substring from a character string.
	TRIM Function (p. 304)	Removes leading characters, trailing characters, or both from a character string.
	UPCASE Function (p. 306)	Converts all letters in an argument to uppercase.
	VERIFY Function (p. 310)	Returns the position of the first character that is unique to an expression.
Date and Time	CURRENT_DATE Function (p. 203)	Returns the current date for the time zone.
	CURRENT_TIME Function (p. 204)	Returns the current time for your time zone.
	CURRENT_TIMESTAMP Function (p. 205)	Returns the date and time for your time zone.
	DATDIF Function (p. 207)	Returns the number of days between two dates after computing the difference between the dates according to specified day count conventions.
	DATE Function (p. 209)	Returns the current date as a SAS date value.

Category	Language Elements	Description
	DATEJUL Function (p. 210)	Converts a Julian date to a SAS date value.
	DATEPART Function (p. 211)	Returns the date as year, month, and day.
	DATETIME Function (p. 212)	Returns the current date and time of day as a SAS datetime value.
	DAY Function (p. 213)	Returns the numeric day of the month from a date or datetime value.
	DHMS Function (p. 214)	Returns a SAS datetime value from date, hour, minute, and second values.
	HMS Function (p. 224)	Returns a SAS time value from hour, minute, and second values.
	HOUR Function (p. 225)	Returns the hour from a time or datetime value.
	INTCK Function (p. 228)	Returns the number of interval boundaries of a given kind that lie between two SAS dates, times, or timestamp values encoded as DOUBLE.
	INTNX Function (p. 234)	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.
	JULDATE Function (p. 242)	Returns the Julian date from a SAS date value.
	MDY Function (p. 250)	Returns a SAS date value from month, day, and year values.
	MINUTE Function (p. 254)	Returns the minute from a time or datetime value.
	MONTH Function (p. 256)	Returns the numeric month from a date or datetime value.
	QTR Function (p. 276)	Returns the quarter of the year from a SAS date value.
	SECOND Function (p. 287)	Returns the second from a time or datetime value.
	TIMEPART Function (p. 301)	Returns the time as hours, minutes, and seconds.
	TODAY Function (p. 303)	Returns the current date as a numeric SAS date value.
	WEEK Function (p. 311)	Returns the week-number value.
	WEEKDAY Function (p. 314)	From a SAS date value, returns an integer that corresponds to the day of the week.
	YEAR Function (p. 315)	Returns the year from a date or datetime value.
	YYQ Function (p. 316)	Returns a SAS date value from year and quarter year values.
Descriptive Statistics	AVG Function (p. 190)	Returns the average of all values in a column.
	CSS Function (p. 202)	Returns the corrected sum of squares of all values in an expression.

Category	Language Elements	Description
	CV Function (p. 206)	Returns the coefficient of variation.
	KURTOSIS Function (p. 243)	Returns the kurtosis of all values in an expression.
	MAX Function (p. 248)	Returns the maximum value in a column.
	MEAN Function (p. 251)	Returns the arithmetic mean (average) of the non-null or nonmissing arguments.
	MEDIAN Function (p. 252)	Returns the median value.
	MIN Function (p. 253)	Returns the minimum value in an expression.
	N Function (p. 259)	Returns the number of non-null or nonmissing numeric values.
	PROBT Function (p. 273)	Returns the probability from a t distribution of the values in an expression.
	RANGE Function (p. 277)	Returns the range between values in an expression.
	SKEWNESS Function (p. 290)	Returns the skewness of all values in an expression.
	STD Function (p. 292)	Returns the standard deviation.
	STDDEV Function (p. 293)	Returns the statistical standard deviation of all values in an expression.
	STDERR Function (p. 294)	Returns the statistical standard error of all values in an expression.
	STUDENTS_T Function (p. 295)	Returns the Student's t distribution of the values in an expression.
	SUM Function (p. 298)	Returns the sum of all the values in an expression.
	USS Function (p. 307)	Returns the uncorrected sum of squares of all the values in an expression.
	VAR Function (p. 308)	Returns the variance.
	VARIANCE Function (p. 309)	Returns the measure of the dispersion of all values in an expression.
Financial	COMPOUND Function (p. 196)	Returns compound interest parameters.
	INTRR Function (p. 239)	Returns the internal rate of return as a decimal value.
	IRR Function (p. 241)	Returns the internal rate of return as a percentage.
	MORT Function (p. 258)	Returns amortization parameters.
	NETPV Function (p. 260)	Returns the net present value as a percent.

Category	Language Elements	Description
	NPV Function (p. 263)	Returns the net present value with the rate expressed as a percentage.
	SAVINGS Function (p. 283)	Returns the balance of a periodic savings by using variable interest rates.
Mathematical	ABS Function (p. 185)	Returns the absolute value of a numeric value expression.
	BETA Function (p. 191)	Returns the value of the beta function.
	DIGAMMA Function (p. 215)	Returns the value of the digamma function.
	ERF Function (p. 216)	Returns the value of the (normal) error function.
	ERFC Function (p. 217)	Returns the value of the complementary (normal) error function.
	EXP Function (p. 218)	Returns the value of the e constant raised to a specified power.
	GAMMA Function (p. 223)	Returns the value of the gamma function.
	LGAMMA Function (p. 244)	Returns the natural logarithm of the Gamma function.
	LOG Function (p. 245)	Returns the natural logarithm (base e) of a numeric value expression.
	LOG2 Function (p. 246)	Returns the base-2 logarithm of a numeric value expression.
	LOG10 Function (p. 247)	Returns the base-10 logarithm of a numeric value expression.
	MOD Function (p. 255)	Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid most unexpected floating-point results.
	SIGN Function (p. 288)	Returns a number that indicates the sign of a numeric value expression.
	SQRT Function (p. 292)	Returns the square root of a value.
Probability	POISSON Function (p. 264)	Returns the probability from a Poisson distribution.
	PROBBETA Function (p. 265)	Returns the probability from a beta distribution.
	PROBBNML Function (p. 266)	Returns the probability from a binomial distribution.
	PROBCHI Function (p. 267)	Returns the probability from a chi-square distribution.
	PROBF Function (p. 268)	Returns the probability from an F distribution.
	PROBGAM Function (p. 269)	Returns the probability from a gamma distribution.
	PROBHYPR Function (p. 270)	Returns the probability from a hypergeometric distribution.
	PROBNEGB Function (p. 272)	Returns the probability from a negative binomial distribution.

Category	Language Elements	Description
	PROBNORM Function (p. 273)	Returns the probability from the standard normal distribution.
Quantile	BETAINV Function (p. 192)	Returns a quantile from the beta distribution.
	GAMINV Function (p. 222)	Returns a quantile from the gamma distribution.
	PROBIT Function (p. 271)	Returns a quantile from the standard normal distribution.
	TINV Function (p. 302)	Returns a quantile from the t distribution.
Special	PUT Function (p. 275)	Returns a value using a specified format.
Trigonometric	ARCOS Function (p. 185)	Returns the arccosine in radians.
	ARSIN Function (p. 186)	Returns the arcsine in radians.
	ATAN Function (p. 187)	Returns the arctangent in radians.
	ATAN2 Function (p. 189)	Returns the arctangent of the x and y coordinates of a right triangle, in radians.
	COS Function (p. 199)	Returns the cosine in radians.
	COSH Function (p. 200)	Returns the hyperbolic cosine in radians.
	SIN Function (p. 289)	Returns the trigonometric sine.
	SINH Function (p. 289)	Returns the hyperbolic sine.
	TAN Function (p. 299)	Returns the tangent.
	TANH Function (p. 300)	Returns the hyperbolic tangent.
Truncation	CEIL Function (p. 194)	Returns the smallest integer greater than or equal to a numeric value expression.
	CEILZ Function (p. 195)	Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing.
	FLOOR Function (p. 219)	Returns the largest integer less than or equal to a numeric value expression.
	FLOORZ Function (p. 220)	Returns the largest integer that is less than or equal to the argument, using zero fuzzing.
	FUZZ Function (p. 221)	Returns the nearest integer if the argument is within 1E-12 of that integer.
	ROUND Function (p. 280)	Rounds the first argument to the nearest multiple of the second argument, or to the nearest integer when the second argument is omitted.

Category	Language Elements	Description
	TRUNC Function (p. 305)	Truncates a numeric value to a specified length.

Dictionary

ABS Function

Returns the absolute value of a numeric value expression.

Category: Mathematical

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 DOUBLE

See "<sql-expression>" on page 58

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`

ARCOS Function

Returns the arccosine in radians.

Category: Trigonometric

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>" on page 58

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 186
- "COS Function" on page 199
- "SIN Function" on page 289

ARSIN Function

Returns the arcsine in radians.

Category: Trigonometric

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>" on page 58

Example

The following statements illustrate the ARSIN function:

Statements	Results
`select arsin(0);`	`0`
`select arsin(1);`	`1.570796`
`select arsin(-0.5);`	`-0.5236`

See Also

Functions:

- "ARCOS Function" on page 185
- "COS Function" on page 199
- "COSH Function" on page 200
- "SIN Function" on page 289

ATAN Function

Returns the arctangent in radians.

Category: Trigonometric

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>" on page 58

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" in *SAS Viya: DS2 Language Reference*
- "TAN Function" on page 299
- "TANH Function" on page 300

ATAN2 Function

Returns the arctangent of the *x* and *y* coordinates of a right triangle, in radians.

Category: Trigonometric

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>" on page 58

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>" on page 58

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`

Statements	Results
`select atan2(5,-3);`	`2.111216`

See Also

- "Handling of Nonexistent Data" on page 10

Functions:

- "ATAN Function" on page 187
- "TAN Function" on page 299

AVG Function

Returns the average of all values in a column.

Categories: Aggregate
Descriptive Statistics

Alias: MEAN

Returned data type: DOUBLE

Syntax

AVG(*expression*)

Arguments

expression
specifies any valid SQL expression.

Data type	DOUBLE
See	"<sql-expression>" on page 58

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 353

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 298

SELECT Statement Clauses:

- "SELECT Clause" on page 324
- "GROUP BY Clause" on page 335
- "HAVING Clause" on page 336

BETA Function

Returns the value of the beta function.

Category: 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`

BETAINV Function

Returns a quantile from the beta distribution.

Category: 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 265

BYTE Function

Returns one character in the ASCII or the EBCDIC collating sequence.

Category: Character

Returned data type: VARCHAR

Syntax

BYTE(*n*)

Arguments

n

specifies an integer that represents a specific ASCII or EBCDIC character.

Range 0–255

Data type CHAR

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 278

CEIL Function

Returns the smallest integer greater than or equal to a numeric value expression.

Category: Truncation

Returned data type: DOUBLE

Syntax

CEIL(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type	DOUBLE
See	"<sql-expression>" on page 58

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 195
- "FLOOR Function" on page 219
- "FLOORZ Function" on page 220

CEILZ Function

Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing.

Category: 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>" on page 58

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 194
- "FLOOR Function" on page 219
- "FLOORZ Function" on page 220

COMPOUND Function

Returns compound interest parameters.

Category: 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 types INTEGER

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.

Category: Character

Returned data type: CHAR

Syntax

COMPRESS(*character-expression*[, *character-list-expression*])

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
See	"<sql-expression>" on page 58

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
See	"<sql-expression>" on page 58

Details

The COMPRESS function allows null arguments. A null argument is treated as a string that has a length of zero.

Based on the number of arguments, the COMPRESS functions 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 will be 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.

To remove digits and plus or minus signs, you could use the following function call:

```
COMPRESS(source, "1234567890+-");
```

Example

This example shows how to remove characters from a string.

Statements	Results
`select compress('abc','a');`	`bc`

COS Function

Returns the cosine in radians.

Category: Trigonometric

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>" on page 58

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 185
- "COSH Function" on page 200
- "SIN Function" on page 289

COSH Function

Returns the hyperbolic cosine in radians.

Category: Trigonometric

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>" on page 58

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 185
- "COS Function" on page 199
- "SINH Function" on page 289
- "TANH Function" on page 300

COUNT Function

Returns the number of rows retrieved by a SELECT statement for a specified table.

Category: Aggregate

Alias: N

Returned data type: DOUBLE

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>" on page 58

*
returns a count of all rows from the table, including rows that contain null values or SAS missing values.

DISTINCT
returns the number of unique values, excluding null values.

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 356

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 324
- "GROUP BY Clause" on page 335
- "HAVING Clause" on page 336

CSS Function

Returns the corrected sum of squares of all values in an expression.

Categories: Aggregate
Descriptive Statistics

Returned data type: DOUBLE

Syntax

CSS(*expression*)

Arguments

expression
specifies any valid SQL expression.

Data type DOUBLE

See "<sql-expression>" on page 58

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 353

The following statement illustrates the CSS function:

Statements	Results
`select css(density) from densities;`	`211483.1`

See Also

Functions:

- "USS Function" on page 307

SELECT Statement Clauses:

- "SELECT Clause" on page 324
- "GROUP BY Clause" on page 335
- "HAVING Clause" on page 336

CURRENT_DATE Function

Returns the current date for the time zone.

Category: Date and Time

Returned data type: DOUBLE

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 204
- "CURRENT_TIMESTAMP Function" on page 205

CURRENT_LOCALE Function

Returns the five-character name of the current locale.

Category: Character

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.

Category: Date and Time

Alias: LOCALTIME

Returned data type: DOUBLE

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 203
- "CURRENT_TIMESTAMP Function" on page 205

CURRENT_TIMESTAMP Function

Returns the date and time for your time zone.

Category: Date and Time

Alias: LOCALTIMESTAMP

Returned data type: DOUBLE

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 203
- "CURRENT_TIME Function" on page 204

CV Function

Returns the coefficient of variation.

Category: 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>" on page 58

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`

Statements	Results
`select cv(8,9,6,.);`	`19.9242421519819`

DATDIF Function

Returns the number of days between two dates after computing the difference between the dates according to specified day count conventions.

Category: 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 DOUBLE

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 DOUBLE

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.

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, 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, the DATDIF function 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 code creates the input table Test using the DS2 language. In SAS Viya, you can submit DS2 statements to SAS Cloud Analytic Services by using the DS2 procedure with the SESSREF= option. Or, you can use the CAS procedure and specify DS2 statements in the ds2.run action. In Python and Lua, you can submit DS2 statements in the ds2.run action.

```
data test (overwrite=yes);
   method run();
      dcl date sdate edate;
      sdate= date'1978-10-16';
      edate= date'1996-02-16';
      sassdate=to_double(sdate);
      sasedate=to_double(edate);
      end;
enddata;
run;
```

This FedSQL SELECT statement returns the actual number of days between two dates:

```
select datdif(sassdate, sasedate, 'act/act') from test;
```

This statement returns the number of days based on a 30-day month and a 360-day year.

```
select datdif(sassdate, sasedate, '30/360') from test;
```

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.

Category: Date and Time

Alias: TODAY

Returned data type: DOUBLE

Syntax

DATE( )

Without Arguments

The DATE function has no arguments.

Details

The SAS date value returned is the number of days from January 1, 1960 to the current date.

Example

The following statement illustrates the DATE function:

Statements	Results
`select date();`	`19611`

See Also

Functions:

- "TODAY Function" on page 303

DATEJUL Function

Converts a Julian date to a SAS date value.

Category: 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 integer 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>” on page 58

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*.

Example

The following statements illustrate the DATEJUL function:

Statements	Results
`select datejul(11365);`	`18992`

See Also

Functions:

- “JULDATE Function” on page 242

DATEPART Function

Returns the date as year, month, and day.

Category:	Date and Time
Returned data type:	DOUBLE

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`

Statements	Results
`select put(datepart(d), date9.) from myoracle.test;`	`24SEP2013`
`select put(datepart(t), time12.) from myoracle.test;`	`5:26:42`

See Also

Functions:

- "TIMEPART Function" on page 301

DATETIME Function

Returns the current date and time of day as a SAS datetime value.

Category: Date and Time

Returned data type: DOUBLE

Syntax

DATETIME( )

Without Arguments

The DATETIME function does not take any arguments.

Details

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:

Statements	Results
`select datetime();`	`1786459057`
`select put(datetime(),datetime18.);`	`10AUG16:15:03:30`

See Also

Functions:

- "DATE Function" on page 209

DAY Function

Returns the numeric day of the month from a date or datetime value.

Category: Date and Time

Returned data type: DOUBLE

Syntax

DAY(*date* | *datetime*)

Arguments

date

specifies any valid expression that represents a date value.

Data type DOUBLE

See "<sql-expression>" on page 58

datetime

specifies any valid expression that represents a datetime value.

Data type DOUBLE

See "<sql-expression>" on page 58

Example

Table: "CustonLine" on page 352

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

Functions:

- "HOUR Function" on page 225
- "MINUTE Function" on page 254
- "MONTH Function" on page 256
- "SECOND Function" on page 287
- "YEAR Function" on page 315

DHMS Function

Returns a SAS datetime value from date, hour, minute, and second values.

Category: 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>" on page 58

hour

specifies a numeric expression that represents an integer from 1 through 12.

Data type DOUBLE

See "<sql-expression>" on page 58

minute

specifies a numeric expression that represents an integer from 1 through 59.

Data type DOUBLE

See "<sql-expression>" on page 58

second

specifies a numeric expression that represents an integer from 1 through 59.

Data type DOUBLE

See "<sql-expression>" on page 58

Details

The DHMS function returns a numeric value that represents a SAS datetime value. This numeric value can be either positive or negative.

Example

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

Functions:

- "HMS Function" on page 224

DIGAMMA Function

Returns the value of the digamma function.

Category: 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>" on page 58

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`

ERF Function

Returns the value of the (normal) error function.

Categories: Mathematical
Arithmetic

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>" on page 58

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 will take 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 217
- "PROBNORM Function" on page 273

ERFC Function

Returns the value of the complementary (normal) error function.

Categories: Mathematical
Arithmetic

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>" on page 58

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 216

EXP Function

Returns the value of the e constant raised to a specified power.

Category: Mathematical

Returned data type: DOUBLE

Syntax

EXP(*expression*)

Arguments

expression
specifies any valid SQL expression that evaluates to a numeric value.

Data type	DOUBLE
See	"<sql-expression>" on page 58

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.

Example

The following statements illustrate the EXP function:

Statements	Results
`select exp(1.0);`	`2.718282`
`select exp(0);`	`1`

See Also

Functions:

- "LOG Function" on page 245

FLOOR Function

Returns the largest integer less than or equal to a numeric value expression.

Category: Truncation

Returned data type: DOUBLE

Syntax

FLOOR(*expression*)

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression>" on page 58

Details

If the result is a number that does not fit into the range of a DOUBLE, the FLOOR function fails.

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

The following statement illustrates the FLOOR function:

Statements	Results
`select floor(1.95);`	`1`

Statements	Results
`select floor(density) from densities;`	67 306 30 323 20 383 31 309 6 247

See Also

Functions:

- "CEIL Function" on page 194
- "CEILZ Function" on page 195
- "FLOORZ Function" on page 220

FLOORZ Function

Returns the largest integer that is less than or equal to the argument, using zero fuzzing.

Category: 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>" on page 58

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

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 194
- "CEILZ Function" on page 195
- "FLOOR Function" on page 219

FUZZ Function

Returns the nearest integer if the argument is within 1E-12 of that integer.

Category: 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>" on page 58

Details

The FUZZ function returns the nearest integer value if the expression is within 1E-12 of the integer (that is, if the absolute difference between the integer 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.

Category: 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>" on page 58

a

specifies any valid expression that evaluates to a numeric shape parameter.

Range	$a > 0$
Data type	DOUBLE
See	"<sql-expression>" on page 58

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 269

GAMMA Function

Returns the value of the gamma function.

Category: 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>" on page 58

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`

HMS Function

Returns a SAS time value from hour, minute, and second values.

Category: Date and Time

Returned data type: DOUBLE

Syntax

HMS(*hour*, *minute*, *second*)

Arguments

hour

specifies a numeric expression that represents an integer from 1 through 12.

Data type DOUBLE

See "<sql-expression>" on page 58

minute

specifies a numeric expression that represents an integer from 1 through 59.

Data type DOUBLE

See "<sql-expression>" on page 58

second

specifies a numeric expression that represents an integer from 1 through 59.

Data type DOUBLE

See "<sql-expression>" on page 58

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

Functions:

- "DHMS Function" on page 214
- "HOUR Function" on page 225
- "MINUTE Function" on page 254
- "SECOND Function" on page 287

HOUR Function

Returns the hour from a time or datetime value.

Category: Date and Time

Returned data type: DOUBLE

Syntax

HOUR(*time* | *datetime*)

Arguments

time

specifies any valid expression that represents a time value.

Data type DOUBLE

See "<sql-expression>" on page 58

datetime

specifies any valid expression that represents a datetime value.

Data type DOUBLE

See "<sql-expression>" on page 58

Example

Table: "CustonLine" on page 352

The following statement illustrates the HOUR function:

Statements	Results
`select hour(endtime) from custonline;`	10 22 19 12 12 16 15 11 10 9
`select hour(current_time);`	14

See Also

Functions:

- "DAY Function" on page 213
- "MINUTE Function" on page 254
- "MONTH Function" on page 256
- "SECOND Function" on page 287
- "YEAR Function" on page 315

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.

Category: Character

Returned data type: DOUBLE

Syntax

INDEX(*target-expression*, *search-expression*)

Arguments

target-expression
specifies any valid expression that evaluates to a character string.

Data type	CHAR
See	"<sql-expression>" on page 58

search-expression
specifies any valid expression that evaluates to a character string to search for in *target-expression*.

Data type	CHAR
Tip	Enclose a literal string of characters in single quotation marks.
See	"<sql-expression>" on page 58

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.

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 227
- "VERIFY Function" on page 310

INDEXC Function

Searches a character expression for specified characters and returns the position of the first occurrence of any of the characters.

Category:	Character
Returned data type:	DOUBLE

Syntax

INDEXC(*target-expression*, *search-expression*[, …*search-expression*])

Arguments

target-expression
specifies any valid expression that evaluates to a character string that is searched.

Data type CHAR

See "<sql-expression>" on page 58

search-expression
specifies the characters to search for in *target-expression*.

Data type CHAR

See "<sql-expression>" on page 58

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 statement:

Statements	Results
`select indexc('aabb','ab');`	`1`

See Also

Functions:

- "INDEX Function" on page 226

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.

Category: Date and Time

Returned data type: DOUBLE

Syntax

INTCK(*interval*[*multiple*][*.shift-index*], *start-date*, *end-date*[, '*method*'])

INTCK(*start-date*, *end-date*[, '*method*'])

Arguments

***interval*[*multiple*][*.shift-index*]**

specifies a basic or complex interval. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The three parts of the interval name are as follows:

interval

specifies a character constant, a variable, or an expression that contains an interval name such as WEEK, MONTH, or QTR.

Data type CHAR

Note For more information, see "Date and Time Intervals" in *SAS Viya Functions and CALL Routines: Reference*.

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 Viya Functions and CALL Routines: Reference*

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 Viya Functions and CALL Routines: Reference*

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

Details

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.

Intervals by Category

Table 7.1 *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, eleventh, and twenty-first of each month	Ten-day periods	TENDAY4.2	Four ten-day periods starting at the second TENDAY period
	SEMIMONTH	Half-month intervals	First and sixteenth 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-month) intervals	January 1 April 1 July 1 October 1	Months	QTR3.2	Three-month intervals starting on April 1, July 1, October 1, and January 1

Category	Interval	Definition	Default Starting Point	Shift Period	Example	Description
	SEMIYEAR	Semiannual (six-month) intervals	January 1 July 1	Months	SEMIYEAR.3	Six-month intervals, March-August, and September-February
	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. These intervals are ISO 8601 compliant. For more information, see "Date and Time Intervals" in *SAS Viya Functions and CALL Routines: Reference*.

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'2013-05-01');`	5
`select intck('week2.2', date'2013-01-07', date'2013-04-01');`	6

Statements	Results
`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 is 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 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.

In the seventh example, the use of variables for the arguments is illustrated.

See Also

Functions:

- "INTNX Function" on page 234

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.

Category: Date and Time

Returned data type: DOUBLE

Syntax

INTNX(*interval*[*multiple*][*.shift-index*], *start-from*, *increment*[, '*alignment*'])

INTNX(*start-from*, *increment*[, '*alignment*'])

Arguments

***interval*[*multiple*][*.shift-index*]**
specifies a basic or complex interval. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The three parts of the interval name are as follows:

interval

specifies a character constant, a variable, or an expression that contains an interval name such as WEEK, MONTH, or QTR.

Data type CHAR

Note For more information, see "Date and Time Intervals" in *SAS Viya Functions and CALL Routines: Reference*.

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 Viya Functions and CALL Routines: Reference*

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 Viya Functions and CALL Routines: Reference*

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 integer 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 237

Default BEGINNING

Data type CHAR

See "Aligning SAS Date Output within Its Intervals" on page 237

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.

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 231.

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. When you use SAME alignment with YEAR, for example, 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 231.

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. These intervals are ISO 8601 compliant. For more information, see "Date and Time Intervals" in *SAS Viya Functions and CALL Routines: Reference*.

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'), date7.);`	19539 30JUN13
`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 228

INTRR Function

Returns the internal rate of return as a decimal value.

Category: Financial

Returned data type: DOUBLE

Syntax

INTRR(*freq*, *c1, c2*[, … *cn*])

Arguments

freq

specifies 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>
specifies the cash payments.

Requirement	At minimum, two cash payments are required.
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 263.) 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 will be 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 expressed as

```
select intrr(1,-400,100,200,300);
```

The value that is returned is 0.1943770996.

See Also

Functions:

- "IRR Function" on page 241

IRR Function

Returns the internal rate of return as a percentage.

Category: Financial

Returned data type: DOUBLE

Syntax

IRR(*freq*, *c1, c2*[,…*cn*])

Arguments

freq

specifies 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>

specifies 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 as

```
select irr(1,-400,100,200,300);
```

The value that is returned is 19.437709963.

See Also

Functions:

- "INTRR Function" on page 239

JULDATE Function

Returns the Julian date from a SAS date value.

Category: 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>" on page 58

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.

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 210

KURTOSIS Function

Returns the kurtosis of all values in an expression.

Categories: Aggregate
Descriptive Statistics

Returned data type: DOUBLE

Syntax

KURTOSIS(*expression*)

Arguments

expression
specifies any valid SQL expression.

Data type	DOUBLE
See	"<sql-expression>" on page 58

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 356

The following statement illustrates the KURTOSIS function:

Statements	Results
`select kurtosis(AvgLow) from worldtemps;`	`-0.87431`

See Also

Functions:

- "STDDEV Function" on page 293

SELECT Statement Clauses:

- "SELECT Clause" on page 324
- "GROUP BY Clause" on page 335
- "HAVING Clause" on page 336

LGAMMA Function

Returns the natural logarithm of the Gamma function.

Category: 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>" on page 58

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.

Category: Mathematical

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>" on page 58

Example

The following statements illustrate the LOG function:

Statements	Results
`select log(1.0);`	`0`

Statements	Results
`select log(10.0);`	`2.302585`

See Also

Functions:

- "LOG10 Function" on page 247
- "LOG2 Function" on page 246

LOG2 Function

Returns the base-2 logarithm of a numeric value expression.

Category: Mathematical

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>" on page 58

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 245
- "LOG10 Function" on page 247

LOG10 Function

Returns the base-10 logarithm of a numeric value expression.

Category: Mathematical

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>" on page 58

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 245
- "LOG2 Function" on page 246

LOWCASE Function

Converts all letters in a character expression to lowercase.

Category: Character

Alias: LOWER

Returned data type: VARCHAR

Syntax

LOWCASE(*expression*)

Arguments

expression

specifies any valid expression that evaluates to a character string.

Requirement	Literal character expressions must be enclosed in single quotation marks.
Data type	CHAR
See	"<sql-expression>" on page 58

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 306

MAX Function

Returns the maximum value in a column.

Categories: Aggregate
Descriptive Statistics

Returned data type: The same data type as the expression

Syntax

MAX(*expression*)

Arguments

expression
specifies any valid SQL expression.

Data type	All data types are supported.
See	"<sql-expression>" on page 58

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 353

The following statement illustrates the MAX function.

Statements	Results
`select max(population) from densities;`	`34248705`

See Also

Functions:

- "MIN Function" on page 253

SELECT Statement Clauses:

- "SELECT Clause" on page 324
- "GROUP BY Clause" on page 335
- "HAVING Clause" on page 336

MDY Function

Returns a SAS date value from month, day, and year values.

Category: Date and Time

Returned data type: DOUBLE

Syntax

MDY(*month*, *day*, *year*)

Arguments

month

specifies a numeric expression that represents an integer from 1 through 12.

Data type DOUBLE

See "<sql-expression>" on page 58

day

specifies a numeric expression that represents an integer from 1 through 31.

Data type DOUBLE

See "<sql-expression>" on page 58

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>" on page 58

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

Functions:

- "DAY Function" on page 213

- "MONTH Function" on page 256
- "YEAR Function" on page 315

MEAN Function

Returns the arithmetic mean (average) of the non-null or nonmissing arguments.

Category: Descriptive Statistics

Returned data type: DOUBLE

Syntax

MEAN(*expression*[, …*expression*])

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>" on page 58

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:

- "MEDIAN Function" on page 252

MEDIAN Function

Returns the median value.

Category: Descriptive Statistics

Returned data type: DOUBLE

Syntax

MEDIAN(*expression*[, …*expression*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression>" on page 58

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*.

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 251

MIN Function

Returns the minimum value in an expression.

Categories: Aggregate
Descriptive Statistics

Returned data type: The same data type as the expression

Syntax

MIN(*expression*)

Arguments

expression
specifies any valid SQL expression.

Data type	All data types are supported.
See	"<sql-expression>" on page 58

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 353

The following statement illustrates the MIN function.

Statements	Results
`select min(density) from densities;`	`6.154657`

See Also

Functions:

- "MAX Function" on page 248

SELECT Statement Clauses:

- "SELECT Clause" on page 324
- "GROUP BY Clause" on page 335
- "HAVING Clause" on page 336

MINUTE Function

Returns the minute from a time or datetime value.

Category: Date and Time

Returned data type: DOUBLE

Syntax

MINUTE(*time* | *datetime*)

Arguments

time

specifies any valid expression that represents a time value.

Data type	DOUBLE
See	"Overview of Expressions and Predicates" on page 41

datetime

specifies any valid expression that represents a datetime value.

Data type	DOUBLE
See	"Overview of Expressions and Predicates" on page 41

Example

Table: "CustonLine" on page 352

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

Functions:

- "DAY Function" on page 213
- "HOUR Function" on page 225
- "MONTH Function" on page 256
- "SECOND Function" on page 287
- "YEAR Function" on page 315

MOD Function

Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid most unexpected floating-point results.

Category: Mathematical

Returned data type: DOUBLE

Syntax

MOD(*expression-1*, *expression-2*)

Arguments

expression-1

specifies any valid SQL expression that evaluates to a numeric value. This argument specifies the dividend.

Data type	DOUBLE
See	"<sql-expression>" on page 58

expression-2
specifies any valid SQL expression that evaluates to a numeric value. This argument specifies the divisor.

Restriction	*expression-2* cannot be 0
Data type	DOUBLE
See	"<sql-expression>" on page 58

Details

The MOD function returns the remainder from the division of *expression-1* by *expression-2*. 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.

- MOD returns zero if the remainder is very close to zero or very close to the value of the second argument.
- MOD returns a 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.

Example

The following statements illustrate the MOD function:

Statements	Results
`select mod(10,3);`	`1`
`select mod(.3,-.1);`	`0.1`
`select mod(1.7,.1);`	`0.1`
`select mod(.9,.3);`	`5.55E-17`

MONTH Function

Returns the numeric month from a date or datetime value.

Category: Date and Time

Returned data type: DOUBLE

Syntax

MONTH(*date* | *datetime*)

Arguments

date

specifies any valid expression that represents a date value.

Data type	DOUBLE
See	"<sql-expression>" on page 58

datetime

specifies any valid expression that represents a datetime value.

Data type	DOUBLE
See	"<sql-expression>" on page 58

Example

Table: "CustonLine" on page 352

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

Functions:

- "DAY Function" on page 213
- "HOUR Function" on page 225
- "MONTH Function" on page 256
- "SECOND Function" on page 287

- "YEAR Function" on page 315

MORT Function

Returns amortization parameters.

Category: 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>" on page 58

p

specifies any valid expression that evaluates to the periodic payment.

Data type DOUBLE

See "<sql-expression>" on page 58

r

specifies any valid expression that evaluates to the periodic interest rate that is expressed as a fraction.

Data type DOUBLE

See "<sql-expression>" on page 58

n

specifies any valid expression that evaluates to the number of compounding periods.

Range $n \geq 0$

Data type DOUBLE

See "<sql-expression>" on page 58

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 example, 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.

Category: Descriptive Statistics

Returned data type: DOUBLE

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	DOUBLE
See	“<sql-expression>” on page 58

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" in *SAS Viya: DS2 Language Reference*

NETPV Function

Returns the net present value as a percent.

Category: 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 263

NMISS Function

Returns the number of null values or SAS missing values in an expression.

Category: Aggregate

Returned data type: DOUBLE

Syntax

NMISS(*expression*)

Arguments

expression
specifies any valid SQL expression.

Data type	CHAR, DOUBLE
See	"<sql-expression>" on page 58

Details

NMISS indicates the total number of null or 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 356

The following statement illustrates the NMISS function:

Statements	Results
`select nmiss(AvgHigh) from worldtemps;`	1

See Also

SELECT Statement Clauses:

- "SELECT Clause" on page 324
- "GROUP BY Clause" on page 335
- "HAVING Clause" on page 336

NPV Function

Returns the net present value with the rate expressed as a percentage.

Category: 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 260

POISSON Function

Returns the probability from a Poisson distribution.

Category: 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>" on page 58

n

specifies any valid expression that evaluates to a random variable.

Range	$n \geq 0$
Data type	DOUBLE
See	"<sql-expression>" on page 58

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`

PROBBETA Function

Returns the probability from a beta distribution.

Category: Probability

Returned data type: DOUBLE

Syntax

PROBBETA(*x*, *a*, *b*)

Arguments

x

is a numeric random variable.

Range $0 \leq x \leq 1$

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.

Category: 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 an integer number of independent Bernoulli trials parameter.

Range $n > 0$

Data type DOUBLE

m

is an integer number of successes random variable.

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`

PROBCHI Function

Returns the probability from a chi-square distribution.

Category: 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);`	`0.5785813293`

PROBF Function

Returns the probability from an *F* distribution.

Category: 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 $nc \geq 0$

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.

Category: 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.

Category: Probability

Returned data type: DOUBLE

Syntax

PROBHYPR(*N*, *K*, *n*, *x*[, *r*])

Arguments

N

is an integer population size parameter.

Range $N \geq 1$

Data type DOUBLE

K

is an integer number of items in the category of interest parameter.

Range $0 \leq K \leq N$

Data type DOUBLE

n

is an integer sample size parameter.

Range $0 \leq n \leq N$

Data type DOUBLE

x

is an integer random variable.

Range $max(0, K + n - N) \leq x \leq min(K,n)$

r

is a numeric odds ratio parameter.

Range $r \geq 0$

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.

Category: 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`

PROBNEGB Function

Returns the probability from a negative binomial distribution.

Category: 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 an integer number of successes parameter.

Range $n \ge 1$

Data type DOUBLE

m

is a positive integer random variable, the number of failures.

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.

Category: 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

Returned data type: DOUBLE

Syntax

PROBT(*expression*)

Arguments

expression
specifies any valid SQL expression.

Data type	DOUBLE
See	“<sql-expression>” on page 58

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 353

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 295

SELECT Statement Clauses:

- “SELECT Clause” on page 324
- “GROUP BY Clause” on page 335

- "HAVING Clause" on page 336

PUT Function

Returns a value using a specified format.

Category: Special

Returned data type: VARCHAR

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 write 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 66.

Example

Table: "WorldTemps" on page 356

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`

QTR Function

Returns the quarter of the year from a SAS date value.

Category: 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>" on page 58

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.

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 316

RANGE Function

Returns the range between values in an expression.

Categories: Aggregate
Descriptive Statistics

Returned data type: the same type as the expression

Syntax

RANGE(*expression*)

Arguments

expression
specifies any valid SQL expression.

Data type	DOUBLE
See	"<sql-expression>" on page 58

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 356

The following statement illustrates the RANGE function:

Statements	Results
`select range(AvgHigh) from worldtemps;`	27

See Also

SELECT Statement Clauses:

- "SELECT Clause" on page 324
- "GROUP BY Clause" on page 335
- "HAVING Clause" on page 336

RANK Function

Returns the position of a character in the ASCII or EBCDIC collating sequence.

Category: Character

Returned data type: DOUBLE

Syntax

RANK(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a character string.

Data type	CHAR
See	"<sql-expression>" on page 58

Details

The RANK function returns an integer that represents the position of the first character in the character expression.

Example

The following statement illustrates the RANK function:

Statement	Result	
	ASCII	EBCDIC
`select "rank"('A');`	65	193

Note: RANK is a reversed FedSQL word. The RANK function must be specified as a delimited identifier.

See Also

Functions:

- "BYTE Function" on page 193

REPEAT Function

Repeats a character expression.

Category: Character

Returned data type: VARCHAR

Syntax

REPEAT(*expression*, *n*)

Arguments

expression

specifies any valid expression that evaluates to a character string.

Interaction	If *expression* is a CHAR expression, then the REPEAT function returns a CHAR value. The maximum length of a CHAR value is 32767 bytes. If *expression* is a VARCHAR expression, then the REPEAT function returns a VARCHAR value. The maximum length of a VARCHAR value is 2^{**31-1} bytes.
Data type	CHAR, VARCHAR
See	"<sql-expression>" on page 58

n

specifies the number of times to repeat *expression*.

Restriction	*n* must be greater than or equal to 0.
Data type	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.

Category: Character

Returned data type: CHAR

Syntax

REVERSE(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a character string.

Data type CHAR

See "<sql-expression>" on page 58

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`

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.

Category: Truncation

Returned data type: DOUBLE

Syntax

ROUND(*expression* [, *rounding-unit*])

Arguments

expression

specifies to be rounded any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "<sql-expression>" on page 58

rounding-unit

specifies a positive numeric expression that specifies the rounding unit.

Data type DOUBLE

See "<sql-expression>" on page 58

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)     - 1200
select round(1234.56789,10)      - 1230
select round(1234.56789,1)       - 1235;
select round(1234.56789,.1)      - 1234.6;
select round(1234.56789,.01)     - 1234.57;
select round(1234.56789,.001)    - 1234.568;
select round(1234.56789,.0001)   - 1234.5679;
select round(1234.56789,.00001)  - 1234.56789;
select round(1234.56789,.1111)   - 1234.5432;
select round(1234.56789,.11111)  - 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;
```

Returns the value 0 (zero).

Operating Environment Information

---

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.

The example above was executed in a z/OS environment. If you use other operating environments, the results might be slightly different.

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 194
- "CEILZ Function" on page 195
- "FLOOR Function" on page 219
- "FLOORZ Function" on page 220

SAVINGS Function

Returns the balance of a periodic savings by using variable interest rates.

Category: Financial

Returned data type: DOUBLE

Syntax

SAVINGS(*base-date*, *initial-deposit-date*, *deposit-amount*, *deposit-number*, *deposit-interval*, *compounding-interval*, *date*, *rate*[, …*date*, *rate*])

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.

```
8458.79415896917
```

- 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 statement 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.

SCAN Function

Returns the *n*th word from a character expression.

Category: Character

Returned data type: CHAR, VARCHAR

Syntax

SCAN(*expression*, *n* [, *delimiters*])

Arguments

expression
specifies any valid expression that evaluates to a character string.

Data type CHAR, VARCHAR

Note The SCAN function supports UTF-8 only if the *expression* argument is a VARCHAR.

See "<sql-expression>" on page 58

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 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.

Interaction ASCII default delimiters are: blank ! $ % & ( ) * + , – . / ; < |. In environments without the ^ character, SCAN uses the ~ character instead.

Data type CHAR, VARCHAR

See "<sql-expression>" on page 58

Details

Leading delimiters before the first word in the expression do not effect SCAN. If there are two or more contiguous delimiters, SCAN treats them as one.

In DS2, if the SCAN function returns a value to a variable that has not yet been given a length, then that variable is given the length of the first argument. If you need the SCAN function to assign to a variable a value that is different from the length of the first argument, then you should use a DECLARE statement for that variable before the statement that uses the SCAN function.

Example

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`

SECOND Function

Returns the second from a time or datetime value.

Category: Date and Time

Returned data type: DOUBLE

Syntax

SECOND(*time* | *datetime*)

Arguments

time

specifies any valid expression that represents a time value.

Data type DOUBLE

See "<sql-expression>" on page 58

datetime

specifies any valid expression that represents a datetime value.

Data type DOUBLE

See "<sql-expression>" on page 58

Example

Table: "CustonLine" on page 352

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

Functions:

- "DAY Function" on page 213

- "HOUR Function" on page 225
- "MINUTE Function" on page 254
- "MONTH Function" on page 256
- "YEAR Function" on page 315

SIGN Function

Returns a number that indicates the sign of a numeric value expression.

Category: Mathematical

Returned data type: DOUBLE

Syntax

SIGN(*expression*)

Arguments

expression

specifies any valid SQL expression that evaluates to a numeric value.

Data type	DOUBLE
See	"<sql-expression>" on page 58

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.

Category: Trigonometric

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>" on page 58

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 185
- "ARSIN Function" on page 186
- "COS Function" on page 199
- "SINH Function" on page 289

SINH Function

Returns the hyperbolic sine.

Category: Trigonometric

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>" on page 58

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 200
- "SIN Function" on page 289

SKEWNESS Function

Returns the skewness of all values in an expression.

Categories: Aggregate
Descriptive Statistics

Returned data type: DOUBLE

Syntax

SKEWNESS(*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>" on page 58

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 356

The following statement illustrates the SKEWNESS function:

Statements	Results
`select skewness(AvgHigh) from worldtemps;`	`-0.69811`

See Also

Functions:

- "STDDEV Function" on page 293

SELECT Statement Clauses:

- "SELECT Clause" on page 324
- "GROUP BY Clause" on page 335
- "HAVING Clause" on page 336

SQRT Function

Returns the square root of a value.

Category: Mathematical

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>" on page 58

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: Aggregate
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>" on page 58

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

Alias: STD

Returned data type: DOUBLE

Syntax

STDDEV(*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>" on page 58

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

"WorldTemps" on page 356

The following statement illustrates the STDDEV function:

Statements	Results
`select stddev(AvgHigh) from worldtemps;`	`7.811414`

See Also

Functions:

- "STDERR Function" on page 294
- "VARIANCE Function" on page 309

SELECT Statement Clauses:

- "SELECT Clause" on page 324
- "GROUP BY Clause" on page 335
- "HAVING Clause" on page 336

STDERR Function

Returns the statistical standard error of all values in an expression.

Categories: Aggregate
Descriptive Statistics

Returned data type: DOUBLE

Syntax

STDERR(*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

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 356

The following statement illustrates the STDERR function:

Statements	Results
`select stderr(AvgHigh) from worldtemps;`	`2.35523`

See Also

Functions:

- "STDDEV Function" on page 293

SELECT Statement Clauses:

- "SELECT Clause" on page 324
- "GROUP BY Clause" on page 335
- "HAVING Clause" on page 336

STUDENTS_T Function

Returns the Student's t distribution of the values in an expression.

Categories: Aggregate
Descriptive Statistics

Alias: T

Returned data type: DOUBLE

Syntax

STUDENTS_T(*expression*)

Arguments

expression
specifies any valid SQL expression.

Data type	DOUBLE
See	"<sql-expression>" on page 58

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 353

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 273

SELECT Statement Clauses:

- "SELECT Clause" on page 324
- "GROUP BY Clause" on page 335
- "HAVING Clause" on page 336

SUBSTRING Function

Extracts a substring from a character string.

Category: Character

Returned data type: VARCHAR

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

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	DOUBLE

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	DOUBLE
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 351

The following statement illustrates the SUBSTRING function.

Statements	Results

`select substring(word2 from 2 for 3);`	`HER`
	`HIN`
	`ODY`

See Also

Functions:

- "TRIM Function" on page 304

SUM Function

Returns the sum of all the values in an expression.

Categories: Aggregate
Descriptive Statistics

Returned data type: The same data type as the expression

Syntax

SUM(*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	DOUBLE
See	"<sql-expression>" on page 58

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 353

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 324
- "GROUP BY Clause" on page 335
- "HAVING Clause" on page 336

TAN Function

Returns the tangent.

Category: Trigonometric

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>" on page 58

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" in *SAS Viya: DS2 Language Reference*
- "ATAN2 Function" in *SAS Viya: DS2 Language Reference*
- "COS Function" on page 199
- "SIN Function" on page 289
- "TANH Function" on page 300

TANH Function

Returns the hyperbolic tangent.

Category: Trigonometric

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>" on page 58

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
`x=tanh(0);`	`0`
`x=tanh(0.5);`	`0.46211715726`
`select tanh(-0.5);`	`-0.46212`

See Also

Functions:

- "ATAN Function" in *SAS Viya: DS2 Language Reference*
- "ATAN2 Function" on page 189
- "COSH Function" on page 200
- "SINH Function" on page 289
- "TAN Function" on page 299

TIMEPART Function

Returns the time as hours, minutes, and seconds.

Category: Date and Time

Returned data type: DOUBLE

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 211

TINV Function

Returns a quantile from the *t* distribution.

Category: 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>" on page 58

df

specifies any valid expression that evaluates to a numeric degrees of freedom parameter.

Range	$df > 0$
Data type	DOUBLE
See	"<sql-expression>" on page 58

nc

specifies any valid expression that evaluates to a numeric noncentrality parameter.

Data type	DOUBLE
See	"<sql-expression>" on page 58

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);`	`2.91998558035372`
`select tinv(.95, 2.5, 3);`	`11.0338336251942`

See Also

Functions:

- "PROBT Function" in *SAS Viya: DS2 Language Reference*

TODAY Function

Returns the current date as a numeric SAS date value.

Category: 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. To display a meaningful date, you must apply a date format to the output value.

Example

The following statement illustrates the TODAY function:

Statements	Results
`select today();`	`19612`

Statements	Results
`select put(today(),date.);`	`14AUG16`

TRIM Function

Removes leading characters, trailing characters, or both from a character string.

Category: Character

Returned data type: VARCHAR

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

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 351

The following statements illustrate the TRIM function:

Statements	Results

`select trim(word1) from afewwords;`	`*some/`
	`*every*`
	`*no*`

Statements	Results

`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`

TRUNC Function

Truncates a numeric value to a specified length.

Category: 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

length-expression
specifies any valid expression that evaluates to a numeric value.

Range 3–8

Data type DOUBLE

See "<sql-expression>" on page 58

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);`	`3.09999847412109`
`select trunc(3.1,5);`	`3.09999999403953`
`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.

Category: Character

Alias: UPPER

Returned data type: VARCHAR, NVARCHAR

Syntax

UPCASE(*expression*)

Arguments

expression

specifies any valid expression that evaluates to a character string.

Data type CHAR, NCHAR

See "<sql-expression>" on page 58

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 247

USS Function

Returns the uncorrected sum of squares of all the values in an expression.

Categories: Aggregate
Descriptive Statistics

Returned data type: DOUBLE

Syntax

USS(*expression*)

Arguments

expression
specifies any valid SQL expression.

Data type	DOUBLE
See	"<sql-expression>" on page 58

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 353

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 202

SELECT Statement Clauses:

- "SELECT Clause" on page 324
- "GROUP BY Clause" on page 335
- "HAVING Clause" on page 336

VAR Function

Returns the variance.

Category: 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>" on page 58

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

Returned data type: DOUBLE

Syntax

VARIANCE(*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>" on page 58

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 353

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 293

SELECT Statement Clauses:

- "SELECT Clause" on page 324
- "GROUP BY Clause" on page 335
- "HAVING Clause" on page 336

VERIFY Function

Returns the position of the first character that is unique to an expression.

Category: Character

Returned data type: DOUBLE

Syntax

VERIFY(*target-expression*, *search–expression*)

Arguments

target-expression

specifies any valid expression that evaluates to a character string that is to be searched.

Requirement	Literal character strings must be enclosed in single quotation marks.
Data type	CHAR
See	"<sql-expression>" on page 58

search-expression

specifies any valid expression that evaluates to a character string.

Requirement	Literal character strings must be enclosed in single quotation marks.
Data type	NCHAR
See	"<sql-expression>" on page 58

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 statements illustrate the VERIFY function:

Statements	Results
`select verify('abc', 'ab');`	3

See Also

Functions:

- "INDEX Function" on page 226

WEEK Function

Returns the week-number value.

Category: Date and Time

Returned data type: DOUBLE

Syntax

WEEK([*sas-date*], ['*descriptor*'])

Without Arguments

If no arguments are specified, the WEEK function returns the week-number value of the current date.

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.

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 312

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 second, third, or fourth, the preceding days are part of the last week of the preceding year.

See "The V Descriptor" on page 313

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.

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 313

Default U

Data type CHAR

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(date'2016-01-01', '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 second, third, or fourth, the preceding days are part of the last week of the preceding year. In the following example, 01jan2016 and 31dec2015 occur in the same week. The first day (Monday) of that week is 28dec2015. Therefore, **`week(date'2016-01-01', 'v')`** and **`week(date'2015-12-31', 'v')`** both return a value of 53. This means that both dates occur in week 53 of the year 2015.

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(date'2016-01-04', '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 example shows the values of the U, V, and W descriptors for the date August 16, 2013.

```
select week(date'2013-08-16', 'u');
select week(date'2013-08-16', 'v');
select week(date'2013-08-16', 'w');
```

The following lines are written to the SAS log.

```
32
33
32
```

See Also

Functions:

- "INTNX Function" on page 234

Formats:

- "WEEKDATE Format" in *SAS Viya: DS2 Language Reference*
- "WEEKDATX Format" in *SAS Viya: DS2 Language Reference*
- "WEEKDAY Format" in *SAS Viya: DS2 Language Reference*

WEEKDAY Function

From a SAS date value, returns an integer that corresponds to the day of the week.

Category: 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>" on page 58

Details

The WEEKDAY function produces an integer that represents the day of the week, where 1 = Sunday, 2 = Monday, …, 7 = Saturday.

Example

The following statement illustrates the WEEKDAY function when the current day is Sunday:

Statements	Results
`select weekday(today());`	1

YEAR Function

Returns the year from a date or datetime value.

Category: Date and Time

Returned data type: DOUBLE

Syntax

YEAR(*date* | *datetime*)

Arguments

date

specifies any valid expression that represents a date value.

Data type DOUBLE

See "<sql-expression>" on page 58

datetime

specifies any valid expression that represents a datetime value.

Data type DOUBLE

See "<sql-expression>" on page 58

Example

Table: "CustonLine" on page 352

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

Functions:

- "DAY Function" on page 213
- "HOUR Function" on page 225
- "MINUTE Function" on page 254
- "MONTH Function" on page 256
- "SECOND Function" on page 287

YYQ Function

Returns a SAS date value from year and quarter year values.

Category: 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 integer that represents the year.

Interaction	The YEARCUTOFF= system option defines the year value for two-digit dates.
Data type	DOUBLE
See	"<sql-expression>" on page 58

quarter
specifies the quarter of the year (1, 2, 3, or 4).

Data type DOUBLE

See "<sql-expression>" on page 58

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

Functions:

- "QTR Function" on page 276
- "YEAR Function" in *SAS Viya: DS2 Language Reference*

Chapter 8

FedSQL Statements

Dictionary

CREATE TABLE Statement

Creates a new table from one or more existing tables.

Restriction: You cannot overwrite an existing table with FedSQL. You must first drop the existing table by using the DROP TABLE statement or some other CAS action that drops tables and then re-create the table with the CREATE TABLE statement. Or you can specify the REPLACE= table option in the CREATE TABLE statement. The REPLACE= table option performs an internal DROP TABLE operation before beginning the CREATE TABLE operation.

Note: 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 ({ }).

Syntax

CREATE TABLE *table*
[\{OPTIONS *SAS-table-option=value*
[…*SAS-table-option=value*] \}]
AS *query-expression*
;

Arguments

table
specifies the name of a table to create.

{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 Chapter 9, "FedSQL Table Options," on page 345

AS *query-expression*
specifies to create a new table from one or more existing tables by selecting rows from the existing tables 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.

See "Creating and Populating Tables from a Query Expression" on page 320

"Overview of FedSQL Expressions and Subqueries" on page 33

"SELECT Statement" on page 321

Details

Overview of the CREATE TABLE Statement

The CREATE TABLE statement enables you to create a table by selecting columns from one or more existing tables using a query expression. In its initial release on CAS, the FedSQL language supports the creation of CAS output tables from CAS input tables only. A *CAS table* is an in-memory table that was created or loaded onto the CAS server by using the SAS procedures or actions that are provided to create and load data into SAS Cloud Analytic Services.

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. The number of columns in the CREATE TABLE statement equal the number of columns that are returned by the SELECT statement. If no column names are specified in the CREATE TABLE statement, the columns and default values that are returned by the SELECT statement are used in the new 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';
```

The output table preserves any column labels and formats that were defined on the input tables. FedSQL does not propagate table labels from input tables to the CREATE TABLE AS output table. Use the LABEL= table option to assign a label to an output table.

See Also

Statements

- "DROP TABLE Statement" on page 321

Table Options

- "LABEL= Table Option" on page 345
- "REPLACE= Table Option" on page 347

DROP TABLE Statement

Removes an in-memory table from the CAS session.

Category: Data Definition

Syntax

DROP TABLE *table* [FORCE];

Arguments

table
specifies the name of the table to be removed. If the table exists in the active caslib, use a one-part table name to identify the table. For tables that exist outside of the active caslib, use a two-part name in the form *caslib.table-name*.

FORCE
specifies that the table is dropped without error processing. Use the FORCE keyword only when you are certain that dropping the table without error processing is what you want to do.

Details

The DROP TABLE statement removes an in-memory table from a CAS session. You might want to drop a table from a CAS session in order to remove a table that is no longer useful, to create a replacement table of the same name, or to reclaim memory.

CAS output tables created with FedSQL exist for the duration of the CAS session, unless you save or promote the tables. DROP TABLE can remove a table that was previously promoted in the caslib. The DROP TABLE statement cannot be used to remove a CAS table that is saved to disk.

See Also

Statements:

- "CREATE TABLE Statement" on page 319

SELECT Statement

Retrieves columns and rows of data from tables.

Categories: Data Definition
Data Manipulation

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>}
<query-specification>::=
    SELECT [ALL | DISTINCT] <select-list> <table-expression>
<select-list>::=
    *
    | column [AS column-alias]
    | expression [AS column-alias]
    | table.*
    | table-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]
    | (<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 324
- "FROM Clause" on page 325
- "WHERE Clause" on page 335
- "GROUP BY Clause" on page 335
- "HAVING Clause" on page 336
- "ORDER BY Clause" on page 337
- "LIMIT Clause" on page 341
- "OFFSET Clause" on page 342

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. Here is an 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.

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 table.

Syntax

SELECT [ALL | DISTINCT] <select-list>

<select-list>::=

*

| *column* [AS *column-alias*]

| <sql-expression> [AS *column-alias*]

| *table*.*

| *table-alias*.*

| <query-specification>

Arguments

ALL

includes all rows, including duplicate rows in the result table.

DISTINCT

eliminates duplicate rows in the result table.

<select-list>

specifies the columns to be selected for the result table.

*

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 "Subqueries" on page 34

<sql-expression> [AS *column-alias*]
derives a column name from an expression.

See "<sql-expression>" on page 58

table*.
selects all columns in the table.

table-alias*.
selects all columns in the table.

See "Table Aliases" on page 327

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 table 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 usually one word. Multiple words can be used if they are quoted. Here is an example:

```
select x as "two words" from table1;
```

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*]

| (<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

CROSS JOIN
defines a join that is the Cartesian product of two tables.

See "Cross Joins" on page 330

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 330

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 334

(<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 "Subqueries" on page 34

table
specifies the name of a table.

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 table.

See "<search-condition>" on page 342

USING (*column* [,...*column*])
specifies which columns to use in an inner or outer join.

See "ON and USING Clauses" on page 333

Overview

The FROM clause enables you to specify source tables. You can reference tables by specifying their table name, 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. 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 table 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 "Overview of Joins" on page 15.

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 table. 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 327. 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 1, "Tables Used in Examples," on page 351.

```
/* FedSQL code for simple join */
proc fedsql;
   title 'Simple Join - GrainProducts and Sales';
   select * from grainproducts, sales;
quit;
```

Output 8.1 *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* 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 table 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 8.2 on page 330.

```
/* FedSQL code for equijoin */
proc fedsql;
   title 'Equijoin - GrainProducts and Sales';
   select * from grainproducts, sales
      where grainproducts.prodid=sales.prodid;
quit;
```

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 8.2 *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.

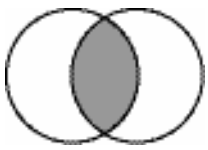
Qualified Joins

Qualified joins provide an easier way to control which rows appear in the result table. You can also further subset the result table with the ON or USING clause.

The two types of qualified joins are inner and outer.

Inner Joins

Figure 8.1 *Inner Join Diagram*



An *inner join* returns a result table that lists 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 table shown in Output 8.3 on page 331.

```
proc fedsql;
   title 'Inner Join - GrainProducts and Sales';
   select *
      from grainproducts inner join sales
         on grainproducts.prodid=sales.prodid;
quit;
```

Output 8.3 *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;
```

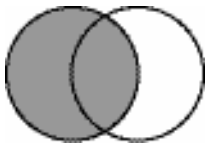
Outer Joins

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 table 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 8.2 *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.

```
title 'Left Outer Join - GrainProducts and Sales';
select *
   from grainproducts left outer join sales
      on grainproducts.prodid=sales.prodid;
```

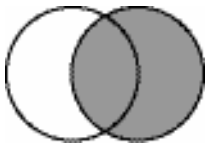
Output 8.4 *Left Outer Join - GrainProducts and Sales Table*

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 8.3 *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.

```
title 'Right Outer Join - GrainProducts and Sales';
select *
   from grainproducts right outer join sales
      on grainproducts.prodid=sales.prodid;
```

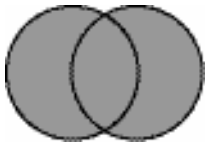
Output 8.5 *Right Outer Join - GrainProducts and Sales Table*

Right 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
.		3975	5	$899,453	Argentina

Full Outer Joins

Figure 8.4 *Full Outer Join Diagram*



A full outer join combines the left outer join and the right outer join. The result table 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.

```
title 'Full Outer Join - GrainProducts and Sales';
select *
   from grainproducts full outer join sales
      on grainproducts.prodid=sales.prodid;
```

Output 8.6 *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 table.

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.

```
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 8.7 *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.

```
title 'Natural Left Outer Join - GrainProducts and Sales';
select *
   from grainproducts natural left outer join sales;
```

Output 8.8 *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 table.

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 table 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 342

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 table. When a condition is met (that is, the condition resolves to true), those rows are displayed in the result table. 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>" on page 58

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 table. If you do not specify the ORDER BY clause, groups returned by the GROUP BY clause are not in any particular order.

Note: FedSQL does not support remerging of summary statistics.

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 342

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 table. 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 8.1 *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 table.

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>" on page 58

ASC
orders the data in ascending order. This is the default 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=
specifies 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 8.2 *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 8.3 *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 Viya National Language Support: Reference Guide*

NUMERIC_COLLATION=

orders integer values within the text by the numeric value instead of characters used to represent the numbers.

Table 8.4 *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 8.5 *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 case there is no difference at levels 1–4.	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.

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 License - ICU 1.8.1 and later" on page 357

The section on Linguistic Collation in *SAS Viya National Language Support: 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 table 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 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.

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.

<search-condition>

Description

Is a combination of one or more operators and predicates that specifies which rows are chosen for inclusion in the result table.

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 8.6 *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 8.7 *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 8.8 *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>" on page 58

Details

The search condition specifies which rows are returned in a result table 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 the parentheses.

Chapter 9

FedSQL Table Options

Overview of Statement Table Options

About FedSQL Statement Table Options

FedSQL statement table options specify actions that affect the processing of a table. They apply only to the table with which they appear.

Restrictions

The availability and behavior of FedSQL statement options are data-source specific. Table options that FedSQL supports for a Base SAS data set or Oracle table are not supported for a CAS table.

Dictionary

LABEL= Table Option

Specifies a label for an output table.

Category: Table Control

Syntax

LABEL=[' | "]*string*[' | "]

Arguments

' *string*'
specifies a quoted text string of up to 256 characters. The string can be enclosed in single or double quotation marks.

Requirements When used in the FedSQL.execDirect action, the LABEL= string must use a different quotation style than the QUERY= string. Single-quotation marks ('), double-quotation marks ("), and double single (' ') quotation marks are all supported for the LABEL= string. Any internal quotation marks must use yet a different quotation style.

In PROC FEDSQL, any internal quotations must use a different quotation style than the outer string. Single-quotation marks ('), double-quotation marks ("), and double single (' ') quotation marks are all supported for the internal quotation.

Details

The labels specified with the LABEL= table option are stored as part of the table's metadata; however, the information is not used in the FedSQL environment. That is, once stored, the label cannot be displayed with FedSQL. In SAS Viya, the label can be viewed by using the CASUTIL procedure with the CONTENTS statement, or by using the CAS procedure with the Tables.tableInfo action. The Tables.tableInfo action is used in Python and Lua.

A label specified for an output table remains a part of the in-memory table for the duration of the CAS session. If the in-memory table is saved or promoted, the label is preserved.

You cannot modify a CAS table with FedSQL. To remove a label from an in-memory table, you must create a new copy of the table with the Label= attribute removed.

Example

These examples assign labels to a FedSQL output table using SAS Viya. They assume that table DemoTable is already loaded into CAS.

```
/* Add a label with PROC CAS */
proc cas;
  fedsql.execdirect result=r status=s query="create table mycars
   {option replace=true
     label='Label test'} as
  select * from demotable";
quit;

/* Add a label with an internal quotation with PROC CAS */
proc cas;
  fedsql.execdirect result=r status=s query='create table mycars
   {option replace=true
     label="Label test with ''internal quotation'' "} as
  select * from demotable';
quit;

/* Add a label with an internal quotation with PROC FEDSQL */
proc fedsql sessref=mysess;
```

```
  create table mycars {option replace=true
  label="Label test with 'internal quotation' "} as
  select * from demotable;
quit;
```

REPLACE= Table Option

Specifies to internally delete an existing table of the same name and create a replacement output table.

Category: Table Control

Default: FALSE

Syntax

REPLACE=[TRUE | FALSE]

Arguments

TRUE
specifies to delete an existing table of the same name and create a replacement output table.

FALSE
specifies to fail the CREATE TABLE operation if a table of the same name already exists. To create a replacement table, you must first use the DROP TABLE statement (or other CAS action that drops tables) to delete the existing table. Then, use CREATE TABLE to create the replacement table.

Details

By default, FedSQL will not overwrite an existing table. The REPLACE= table option will delete and then re-create an existing table of the same name when set to TRUE. If the output table exists and the REPLACE= table option is set to FALSE (the default value), an error will occur because the existing table will not be deleted.

Part 3

Appendixes

Appendix 1

Tables Used in Examples

AfewWords

The column Word was created with a data type of varchar(10).

Table A1.1 *AfewWords*

Word1	Word2
*some/	WHERE
every	THING
no	BODY

Customers

Table A1.2 *Customers*

Custid	Name	Address	City	State	Country	Phone	Initial Order
1	Peter Frank	300 Rock Lane	Boulder	CO	United States	3039564321	20120114
2	Jim Stewart	1500 Lapis Lane	Little Rock	AR	United States	8705553978	20120320
3	Janet Chien	75 Jujitsu	Nagasaki		Japan	01181956879932	20120607
4	Qing Ziao	10111 Karaje	Tokyo		Japan	0118136774351	20121012
5	Humberto Sertu	876 Avenida Blanca	Buenos Aires		Argentina	01154118435029	20121215

CustonLine

Table A1.3 *CustonLine*

Customer Number	BeginTime	EndTime
US-C-37533944	01SEP2013:10:00:00.000	01SEP2013:10:05:01.253
GB-W-33944332	02OCT2013:22:15:33.000	02OCT2013:22:21:09.421
SP-M-29443992	15OCT2013:18:44:25.000	15OCT2013:19:04:55.746
US-A-37144324	01NOV2013:12:03:59.000	01NOV2013:12:25:09.398
FR-P-98384488	01DEC2013:12:15:34.000	01DEC2013:12:47:45.221
GB-L-24995559	02JAN2013:15:43:24.000	02JAN2013:16:06:15.766
FR-L-42339887	16JAN2013:14:55:00.000	16JAN2013:15:05:56.288
GB-P-87559899	01FEB2013:11:02:44.000	01FEB2013:11:15:33.955
SP-N-44333958	01MAR2013:10:14:33.000	01MAR2013:10:35:27.908

Customer Number	BeginTime	EndTime
GB-R-24994990	15MAR2013:09:00:06.000	15MAR2013:09:06:20.475

Densities

Table A1.4 *Densities*

Name	Population	SquareMiles	Density
Afghanistan	17,070,323	251825	67.79
Albania	3,407,400	11100	306.97
Algeria	28,171,132	919595	30.63
Andorra	64,634	200	323.17
Angola	9,901,050	481300	20.57
Antigua and Bar	65,644	171	383.88
Argentina	34,248,705	1073518	31.90
Armenia	3,556,864	11500	309.29
Australia	18,255,944	2966200	6.15
Austria	8,033,746	32400	247.96

Employees

EmpID	Dept	Emp_Name	Pos	Hire_Date
1	10	Jim Barnes	Manager	2000-11-26
2	20	Clifford James	Manager	2000-11-26
3	30	Barbara Sandman	Manager	2000-11-26
4	40	William Baylor	Manager	2000-11-26

EmpID	Dept	Emp_Name	Pos	Hire_Date
5	20	Greg Welty	Developer	2004-11-26
6	20	Penny Jackson	Developer	2004-11-26
7	10	Edward Murray	Sales Associate	2001-11-26
8	10	Ronald Thomas	Sales Associate	2002-11-26
9	30	Elsie Marks	Executive Assistant	2002-02-11
10	40	Bruno Kramer	Grounds support technician	2003-11-02

GrainProducts

Table A1.5 *GrainProducts*

Prodid	Product
1424	Rice
3421	Corn
3234	Wheat
3485	Oat

Products

Table A1.6 *Products*

Prodid	Product
3234	Rice
1424	Corn
3421	Wheat
3422	Oat
3975	Barley

Sales

Table A1.7 Sales

Prodid	Custid	Totals	Country
3234	1	189400	United States
1424	3	555789	Japan
3421	4	781183	Japan
3421	2	2789654	United States
3975	5	899453	Argentina

WorldCityCoords

Table A1.8 WorldCityCoords

City	Country	Latitude	Longitude
Algiers	Algeria	37	3
Shanghai	China	31	121
Hong Kong	Hong Kong	22	114
Bombay	India	19	73
Calcutta	India	22	88
Amsterdam	Netherlands	52	5
Lagos	Nigeria	6	3
Madrid	Spain	40	.
Zurich	Switzerland	47	8
Caracas	Venezuel	10	-67
	China	40	116

WorldTemps

Table A1.9 *WorldTemps*

City	Country	AvgHigh	AvgLow
Algiers	Algeria	90	45
Amsterdam	Netherlands	70	33
Beijing	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 2

ICU License Agreement

Recommended Reading

- *SAS Cloud Analytic Services: Fundamentals*
- *SAS Cloud Analytic Services: CAS Procedure Programming Guide and Reference*
- *SAS Cloud Analytic Services: Getting Started with Python*
- *SAS Cloud Analytic Services: System Programming Guide*
- *SAS Cloud Analytic Services: Authorization*
- *SAS Viya Data Management and Utility Procedures Guide*

For a complete list of SAS publications, go to sas.com/store/books. If you have questions about which titles you need, please contact a SAS Representative:

SAS Books
SAS Campus Drive
Cary, NC 27513-2414
Phone: 1-800-727-0025
Fax: 1-919-677-4444
Email: sasbook@sas.com
Web address: sas.com/store/books

Index