SAS® Viya™ 3.3: FedSQL Programming for SAS® Cloud Analytic Services

SAS® Documentation
November 7, 2017

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2017. *SAS® Viya™ 3.3: FedSQL Programming for SAS® Cloud Analytic Services*. Cary, NC: SAS Institute Inc.

SAS® Viya™ 3.3: FedSQL Programming for SAS® Cloud Analytic Services

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

December 2017

3.3-P1:casfedsql

Contents

What's New in FedSQL Programming for SAS Cloud Analytic Services

Overview

In SAS Viya 3.3, FedSQL adds several new features:

- an implicit pass-through facility
- an option for controlling the FedSQL query planner
- support for new CAS data types
- two new table options in FedSQL for CAS

FedSQL Implicit Pass-Through Facility for CAS

FedSQL for CAS adds an implicit pass-through facility. In the initial release, the facility provides single-source, full-query implicit pass-through. When a request is accessing a single data source, an attempt is made to implicitly pass the full query down to the data source for processing. If pass-through is not possible, the request is processed locally on the CAS server. To be eligible for implicit pass-through, all of the tables in the FedSQL request must exist in the same caslib, and the tables cannot already have been loaded into the CAS session. For more information, see "FedSQL Implicit Pass-Through Facility in CAS" on page 5.

FedSQL implicit pass-through functionality is supported for Amazon Redshift, DB2, Hadoop (Hive and Impala), ODBC, Oracle, PostgresSQL, SAP Hana, and Teradata data sources.

Controlling the FedSQL Query Planner

A Cntl= option has been added to both PROC FEDSQL and to the fedSql.execDirect action that enables you to specify optional control parameters for the FedSQL query planner in CAS. The following parameters are supported:

disablePassThrough
: disables implicit FedSQL pass-through. Referenced tables are loaded into the CAS server for processing.

preserveJoinOrder
: joins tables in the specified order instead of an order chosen by the FedSQL query optimizer.

requireFullPassThrough
: stops processing the FedSQL request when implicit pass-through of the full query cannot be achieved.

For more information, see "Modifying the Query Plan" on page 7.

CAS Data Types

Beginning with SAS Viya 3.3, FedSQL extends its CAS data type support to include INT64 and INT32 data types in addition to CHAR, DOUBLE, and VARCHAR. For more information, see "Data Types" on page 15 and "Handling of Nonexistent Data" on page 19.

Table Options

The fedSql.execDirect action supports new COMPRESS= and REPLICATION= table options. See "COMPRESS= Table Option" on page 112 and "REPLICATION= Table Option" on page 114.

Part 1

Concepts

Chapter 1

Introduction to SAS FedSQL for CAS

Introduction to the SAS FedSQL Language in CAS

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

In SAS Cloud Analytic Services (CAS), FedSQL provides a scalable, threaded, high-performance way to query data. FedSQL enables you to join data using industry-standard query expressions and SQL expressions. FedSQL can be used to join relational data from multiple data sources in a single request.

For applications, FedSQL provides a common SQL syntax across all data sources. That is, FedSQL is a vendor-neutral SQL dialect that accesses data from various data sources without having to submit queries in the SQL dialect that is specific to the data source. In addition, a single FedSQL query can target data in several data sources and return a single result set.

You can submit FedSQL statements to the CAS server from a SAS or SAS Viya session by using the FEDSQL procedure. For more information about the FEDSQL procedure, see *Base SAS Procedures Guide*. You can also submit FedSQL statements to the CAS server in SAS Viya by using the fedSql.execDirect action. For more information about the fedSql.execDirect action, see *SAS Viya: System Programming Guide*.

When FedSQL statements are executed by the CAS server, the FedSQL result set is always an in-memory CAS table. You can use other CAS actions to persist the result set on the CAS server or to save the result set to an external data source.

Running FedSQL Programs in CAS

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

- Using the FEDSQL procedure. The FEDSQL procedure can execute FedSQL statements in SAS libraries as well as on the CAS server. FedSQL supports a more expanded syntax for SAS libraries than it does for CAS libraries. You must use the CAS FedSQL syntax described in this book for requests submitted to the CAS server. For more information about the FedSQL procedure, see "FEDSQL Procedure" in *Base SAS Procedures Guide*.
- Using the fedSql.execDirect action. The fedSql.execDirect action can be called from a SAS Viya CASL program or from a SAS Viya Python, Lua, or R program. In CASL, the fedSql.execDirect action is used with the CAS procedure. For more information about the CAS procedure, see *SAS Cloud Analytic Services: CASL Reference*. For information about using fedSql.execDirect in the other programming environments, see *Getting Started with SAS Viya for Lua*, *Getting Started with SAS Viya for Python*, and *Getting Started with SAS Viya for R*. For more information about the fedSql.execDirect action, see *SAS Viya: System Programming Guide*.

When you are using the fedSql.execDirect action, FedSQL statements are submitted to the CAS server in a quoted string. A benefit of using PROC FEDSQL to submit FedSQL statements to the CAS server is that your FedSQL statements do not need to be quoted.

A benefit of using the fedSql.execDirect action is that it enables you to take advantage of the functionality of the host language when creating your programs. For example, you might use the host language to build your query string before you call the execDirect action, and then call the action repeatedly in a loop, changing a parameter or part of the query string each time through the loop. Or, you might use the host language to post-process CAS result tables on the client. See "Query a Database Table with the execDirect Action" in *SAS Viya: System Programming Guide* for a simple example of using host language elements with the fedSql.execDirect action.

Supported Statements

The following FedSQL statements are supported in CAS:

- CREATE TABLE, with the AS query expression

- DROP TABLE
- SELECT

For more information about statement functionality, see Chapter 8, "FedSQL Statements," on page 87.

Supported Data Sources

FedSQL statements are executed in CAS by the fedSql.execDirect action. The execDirect action uses SAS Viya Data Connectors to access SAS data and data from third-party data sources with CAS. The data connectors provide single-pass, load-on-demand access to specified tables in CAS when a table name is referenced with a caslib in a FedSQL statement. Data can also be loaded explicitly into a CAS session for processing with execDirect. For a listing of available SAS Viya Data Connectors, see "Quick Reference for Data Connector Syntax" in *SAS Cloud Analytic Services: User's Guide*. Also see "Working with SAS Data Connectors" in *SAS Cloud Analytic Services: User's Guide* and "Working with Caslibs" in *SAS Viya: System Programming Guide*.

Beginning in SAS Viya 3.3, FedSQL supports implicit SQL pass-through for the following data sources through SAS Viya Data Connectors:

- Amazon Redshift
- DB2 (UNIX)
- Hadoop (Hive and Impala)
- ODBC
- Oracle
- PostgreSQL
- SAP Hana
- Teradata (UNIX)

Data access is serial by default. Parallel data access is available for Teradata and Hadoop when SAS Viya Data Connector Accelerator software is installed.

FedSQL Implicit Pass-Through Facility in CAS

Overview

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

IP provides two categories of performance benefits: those that result from reducing data transfer volume and those that result from taking advantage of data-source-specific capabilities. Benefits from the first category come from performing the query on the data source. The number of rows that are transferred from the data source to FedSQL can be significantly reduced, thereby decreasing the overall query processing time. Benefits from the second category, resulting from data-source-specific capabilities such as massively parallel processing, advanced join techniques, data partitioning, table

statistics, and column statistics, depend on the data source. These capabilities often allow the data source to perform the SQL query more quickly than FedSQL.

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

FedSQL in CAS provides single-source, full-query implicit pass-through. When a request is accessing a single data source, an attempt is made to implicitly pass the full query down to the data source. If pass-through is not possible, the request is processed locally on the CAS server.

How to Use the FedSQL Implicit Pass-Through Facility

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

Conditions for Single-Source Pass-Through

Here are the requirements for pass-through:

- The data source must be an SQL data source.
- The tables cannot already have been loaded into the CAS session.
- All tables that are specified in the FedSQL request must exist in the same caslib. Merges and joins of tables that exist in different caslibs are automatically loaded into CAS for processing.
- The query cannot contain an ORDER BY clause.

FedSQL can pass queries implicitly only when the SQL syntax is ANSI-compliant. The following limitations might prevent IP:

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

How FedSQL Runs in CAS

Overview

FedSQL statements that are submitted to the CAS server are processed by the fedSql.execDirect action. The execDirect action uses the FedSQL query optimizer and FedSQL pass-through capabilities to plan and execute queries. It uses CAS to partition and order the data.

A FedSQL query plan is divided into stages. Each stage requires a stand-alone SQL query. The following FedSQL plan nodes are turned into execution stages:

- SeqScan (when it is the root of the plan)

- HashJoin
- MergeJoin
- NestLoop
- Sort
- Group
- Aggregate
- Unique
- Limit
- Result

Each node represents an internal algorithm for processing requests. The following plan nodes currently support threaded execution:

- SeqScan
- MergeJoin
- HashJoin
- NestLoop

In summary, Reads and Joins are processed in parallel, except FULL OUTER joins where the join condition is something other than a simple equality condition on columns.

The following operations are currently processed by a single CAS worker:

- SELECT DISTINCT (UNIQUE execution stage)
- LIMIT and OFFSET (LIMIT execution stage)
- GROUP BY aggregations where one or more group expressions are not simple column references.

Modifying the Query Plan

Beginning with SAS Viya 3.3, FedSQL supports a Cntl option that enables you to control aspects of the query plan. The Cntl option enables you to specify the following instructions:

disablePassThrough=true | false
: The execDirect action attempts to use implicit SQL pass-through for all data sources that support it by default (disablePassThrough=false). Specifying disablePassThrough=true disables implicit pass-through. The data connector loads the target tables into CAS for processing.

preserveJoinOrder=true | false
: The FedSQL query optimizer rewrites queries to optimize join processing by default (preserveJoinOrder=false). When preserveJoinOrder=true is specified, FedSQL joins tables in the specified order instead of an order that is chosen by the FedSQL query optimizer. For example, when preserveJoinOrder=true, FedSQL processes the following request as follows:

```
select * from a, b, c, d where...;
```

- Stage 1 — join tables A and B.
- Stage 2 — join rows from Stage 1 with table C.
- Stage 3 — join rows from Stage 2 with table D.

Note: preserveJoinOrder has no effect on queries that are passed down to the external data source.

requireFullPassThrough=true | false
The execDirect action loads external data into CAS for processing when implicit pass-through cannot be achieved (requireFullPassThrough=false). When requireFullPassThrough=true is specified, FedSQL stops processing the request when implicit pass-through of the full query cannot be achieved. No data is loaded into CAS and no output table or result set is produced.

Note: This instruction requests the opposite behavior of disablePassThrough=yes.

In PROC FEDSQL, Cntl is specified as a procedure option. Instructions are specified within parenthesis. The value **true** is implied by the mention of an instruction. Here is an example of how the instructions are specified in the FEDSQL procedure:

```
proc fedsql sessref="mysess" cntl=(requireFullPassThrough);
...FedSQL statements...;
quit;
```

Multiple instructions are separated by a space.

```
proc fedsql sessref="mysess" cntl=(preserveJoinOrder disablePassThrough);
...FedSQL statements...;
quit;
```

In the fedSql.execDirect action, Cntl is specified as an action parameter. Instructions are specified within braces as name=value pairs. Here is an example:

```
proc cas;
   fedsql.execdirect
     cntl={requireFullPassThrough=true}
     query="...FedSQL statements...";
quit;
```

In the action, multiple instructions can be separated by a space or a comma.

```
proc cas;
   fedsql.execdirect
     cntl={preserverJoinOrder=true, disablePassThrough=true}
     query="...FedSQL statements...";
quit;
```

Viewing the Query Plan

To see the query plan for a given FedSQL request, set the Method option. The Method option generates a text description of the nodes and stages in the query plan for a given request and writes the output to the SAS log.

In PROC FEDSQL, Method is specified as a procedure option. The keyword is preceded by an underscore. Here is an example:

```
proc fedsql sessref="mysess" _method;
  ...FedSQL statements...;
quit;
```

In the fedSql.execDirect action, Method is specified as an action parameter. Here is an example:

```
proc cas;
   fedsql.execdirect
     method=true
```

```
    query="...FedSQL statements...";
quit;
```

Note: The execDirect syntax shown above is specific to CASL. See *SAS Viya: System Programming Guide* for examples that use Python, Lua, and R syntax.

You can also get information about query plan nodes without executing the FedSQL request.

- In PROC FEDSQL, specify the NOEXEC option with the _Method option to get the query plan without executing the query.
- In the fedSql.execDirect action, specify the ValidateOnly option with the Method option to get the query plan without executing the query.

FedSQL Query Walk-Through

Here is an example of a FedSQL query and its query plan.

The FedSQL query:

```
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;
```

The query plan:

```
Methods for full query plan 1
 ----------------------------
 Number of Sorts Performed is : 1
 Number of Joins Performed is : 2
         Sort
           MergeJoin (INNER)
             SubqueryScan
               Agg
                 Sort
                   SeqScan from castera.WORLDTEMPS
             Sort
               HashJoin (INNER)
                 SeqScan from castera.WORLDCITYCOORDS
                 SeqScan from castera.WORLDTEMPS

 Methods for stage 1 2
 --------------------
         Agg
             SeqScan with _pushed_ order by from castera.WORLDTEMPS

         Stage query: create table "castera"."__fedsql_1__"
{options replace=true replication=0 tableID=2} as select "T1"."COUNTRY",
 AVG ("T1"."AVGHIGH")  as "AVGHIGHNATION" from "castera"."WORLDTEMPS"
{options tableID=1} T1 group by "T1"."COUNTRY"

         Number of SQL I/O threads: 32 min, 56 max

 Methods for stage 3 3
 --------------------
         HashJoin (INNER)
           SeqScan from castera.WORLDCITYCOORDS
           SeqScan from castera.WORLDTEMPS

         Stage query: create table "castera"."__fedsql_3__"
{options replace=true replication=0 tableID=3} as select "T2"."AVGHIGH",
 "T2"."COUNTRY", "T1"."CITY", "T1"."COUNTRY" as "COUNTRY_2", "T1"."LATITUDE",
"T1"."LONGITUDE" from "castera"."WORLDCITYCOORDS"
 {options tableID=1} T1 _hash_ inner join "castera"."WORLDTEMPS"
{options REPL=YES tableID=2} T2 on  ("T1"."CITY"="T2"."CITY")

         Number of SQL I/O threads: 32 min, 56 max

 Methods for stage 4 4
 --------------------
         HashJoin (INNER)
           SeqScan from castera.__fedsql_3__
           SeqScan from castera.__fedsql_1__

         Stage query: create table "castera"."__fedsql_4__"
{options replace=true replication=0 tableID=3} as select "T2"."CITY",
 "T2"."COUNTRY_2" as "COUNTRY", "T2"."LATITUDE", "T2"."LONGITUDE",
"T2"."AVGHIGH"
as "AVGHIGHCITY", "T1"."AVGHIGHNATION" from "castera"."__fedsql_3__"
{options tableID=2} T2 _hash_ inner join "castera"."__fedsql_1__"
{options REPL=YES tableID=1} T1 on ("T1"."COUNTRY"="T2"."COUNTRY")

         Number of SQL I/O threads: 32 min, 56 max

 Methods for stage 5 5
 --------------------
         Sort
           SeqScan from castera.__fedsql_4__

         Stage query: select "T1"."CITY", "T1"."COUNTRY", "T1"."LATITUDE",
"T1"."LONGITUDE", "T1"."AVGHIGHCITY",
 "T1"."AVGHIGHNATION" from "castera"."__fedsql_4__" {options REPEAT=YES }
T1 order by 2 collate linguistic (locale=en_US), 1 collate
 linguistic (locale=en_US)

         Number of SQL I/O threads: 1
```

This FedSQL query specifies to join select columns from two CAS tables named WorldCityCoords and WorldTemp (described in Appendix 1, "Tables Used in Examples," on page 119) and adds a calculated column named AvgHighNation to each row of the merged result set. It uses a subquery to create the new column. The tables exist in a Teradata database and are referenced by the caslib CASTERA

1. The query plan begins with a summary of the plan nodes that are used to process the request in the order in which they are executed. It then describes each stage of the plan.
2. This query plan processes the subquery in table WorldTemps first. In Stage 1, FedSQL performs an aggregate sort on column AVGHIGH using the values in column COUNTRY to create a new column named AVG_HIGH_NATION. Temporary table _fedsql_1__ is created to hold the results of the subquery.
3. The query plan then continues to the other columns in the SELECT clause. This step requires no processing. Thus, Stage 2 is omitted from the plan. In Stage 3, the plan selects and joins other specified columns from the WorldTemps and WorldCityCoords tables. It creates a temporary table __fedsql_3__ to hold the results.
4. In Stage 4, the plan joins temporary tables __fedsql_3__ and __fedsql_1__ to create temporary table _fedsql_4_.
5. Finally, stage 5 performs a sort and sequential scan to display the contents of temporary table _fedsql_4_.

The number of threads per worker that is used to process each stage is shown at the end of each stage.

Here is an example of the output from the same request when the ValidateOnly option is specified along with the Method option.

```
Methods for full query plan
---------------------------
Number of Sorts Performed is : 1
Number of Joins Performed is : 2
        Sort
          MergeJoin (INNER)
            SubqueryScan
              Agg
                Sort
                  SeqScan from castera.WORLDTEMPS
            Sort
              HashJoin (INNER)
                SeqScan from castera.WORLDCITYCOORDS
                SeqScan from castera.WORLDTEMPS
```

Executing a FedSQL Request Against Multiple Data Sources in CAS

You can execute a FedSQL request against multiple data sources in CAS by identifying tables using a two-part table name in the form *caslib.table-name*. The tables from the specified caslibs are then loaded into CAS for processing.

Managing FedSQL Output Tables

In CAS, the FedSQL CREATE TABLE statement creates in-memory CAS output tables. The output tables exist for the length of the CAS session only. To persist a table in CAS between sessions, use the table.promote action. To save a table, use the table.save action. The table.save action saves a table to a caslib's data source. For more information, see "Promote table" in *SAS Viya: System Programming Guide* and "Save table" in *SAS Viya: System Programming Guide*.

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, Python, Lua, and R programmers who want 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 Viya: FedSQL Programming 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

Locale

The locale identifies the language and possibly a regional dialect to use for the user interface. The fedSql.execDirect action honors the locale set in the LOCALE= CAS session option for sorting and formatting. The default session locale is en_US. Sort ordering for the execDirect action honors the collation sequence indicated in the COLLATE= CAS session option. The default value, COLLATE="UCA", requests a locale-appropriate collation sequence. COLLATE= also supports an MVA option, which requests SAS client collating. When COLLATE="MVA", execDirect performs binary sort ordering. Sort ordering cannot be changed in the SELECT statement.

Data Types

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

The following table lists the data types that FedSQL supports for CAS. Beginning with SAS Viya 3.3, CAS tables support INT64 and INT32 data types in addition to CHAR, DOUBLE, and VARCHAR.

Table 2.1 *FedSQL Data Type Translation for CAS Tables*

FedSQL Data Type	CAS Data Type	Description
BIGINT	INT64	stores a large signed, exact whole number, with a precision of 19 digits. The range of integers is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Integer data types do not store decimal values; fractional portions are discarded. When FedSQL encounters a value that corresponds to the most negative possible INT64 value, it treats it as a null or nonexistent value.
CHAR(*n*)	CHAR(*n*)	Stores a fixed-length character string, where *n* is the maximum number of characters to store. The maximum 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	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.

FedSQL Data Type	CAS Data Type	Description
INTEGER	INT32	stores a regular size signed, exact whole number, with a precision of 10 digits. The range of integers is -2,147,483,648 to 2,147,483,647. Integer data types do not store decimal values; fractional portions are discarded. When FedSQL encounters a value that corresponds to the most negative possible INT32 value, it treats it as a null or nonexistent value.
VARCHAR(*n*)	VARCHAR(*n*)	Stores a varying-length character string, where *n* is the maximum number of characters to store. Each character uses 1 byte of storage. The maximum number of characters is not required to store each value. If **`varchar(10)`** is specified and the character string is only five characters long, only five characters are stored in the column.

When SAS Viya Data Connectors read DATE, TIME, and TIMESTAMP columns from an ANSI-compliant data source, they convert the columns to data type DOUBLE, with a SAS format applied. The TIME8. SAS format is applied to time values. The DATE9. SAS format is applied to date values. The DATETIME25.6 SAS format is applied to timestamp values.

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 21.

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 DOLLARw.d format, which converts numeric values to a decimal monetary value, writes the numeric value 4503945867 as $4,503,945,867.00.

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 77.

Handling of Nonexistent Data

FedSQL for CAS treats null values in CHAR, DOUBLE, and VARCHAR columns as SAS missing values. That is, when FedSQL reads a null value in a column of type CHAR from a ANSI-compliant data source using the CAS server, it converts the ANSI null value to a SAS character missing value (blank-filled character string). FedSQL converts ANSI null values in columns of type VARCHAR to a SAS character missing value (empty character string). FedSQL converts ANSI null values in columns of type DOUBLE to a SAS numeric missing value (a dot or period).

FedSQL treats null values in INT64, INT32, and DOUBLE columns that were converted from DATE, TIME, and TIMESTAMP data types as ANSI null values.

Processing SAS missing values is different from ANSI handling of null values. In ANSI SQL, nulls and nonexistent data have no data value. That is, nulls are treated as unknown values. In SAS, they are treated as known values. The use of missing values has implications for query processing, particularly in a WHERE clause, HAVING clause, or an outer join ON clause.

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

As an illustration, consider this example, which creates table NullTest in CAS that contains a column of each CAS data type and stores a null value in it. The example then submits two FedSQL requests with WHERE clauses to find null values. The first

WHERE clause is an equality test. It searches for values “equal to” null in each column. The second clause looks for the existence of a null value in each column. To avoid duplication, only the log outputs of the FedSQL queries are shown.

```
proc ds2 sessref=mysess;
data nulltest;
dcl bigint bigCol;
dcl int intCol;
dcl double doubleCol;
dcl varchar vcCol;
dcl char charCol;
dcl date dateCol;
dcl time timeCol;
dcl timestamp tsCol;
method init();
  bigCol=NULL;
  intCol=NULL;
  doubleCol=NULL;
  vcCol=NULL;
  charCol=NULL;
  dateCol=NULL;
  timeCol=NULL;
  tsCol=NULL;
end; enddata; run; quit;
```

Log 2.1 *Log from the Equality Tests*

```
1          OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
 72
 73         proc fedsql sessref=mysess;
 74         select * from nulltest WHERE bigCol = NULL;
 NOTE: No rows returned.
 75         select * from nulltest WHERE intCol = NULL;
 NOTE: No rows returned.
 76         select * from nulltest WHERE doubleCol = NULL;
 77         select * from nulltest WHERE vcCol = NULL;

 78         select * from nulltest WHERE charCol = NULL;
 79         select * from nulltest WHERE dateCol = NULL;
 NOTE: No rows returned.
 80         select * from nulltest WHERE timeCol = NULL;
 NOTE: No rows returned.
 81         select * from nulltest WHERE tsCol = NULL;
 NOTE: No rows returned.
 82         quit;
```

Log 2.2 *Log from the Existence Tests*

```
1          OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
 72
 73         proc fedsql sessref=mysess;
 74         select * from nulltest WHERE bigCol IS NULL;
 75         select * from nulltest WHERE intCol IS NULL;
 76         select * from nulltest WHERE doubleCol IS NULL;
 77         select * from nulltest WHERE vcCol IS NULL;
 78         select * from nulltest WHERE charCol IS NULL;
 79         select * from nulltest WHERE dateCol IS NULL;
 80         select * from nulltest WHERE timeCol IS NULL;
 81         select * from nulltest WHERE tsCol IS NULL;
 82         quit;
```

The equality tests fail to return rows for columns of type BIGINT, INTEGER, DATE, TIME, and TIMESTAMP. The null values for these data types are not “known” to the CAS server. The existence tests were all successful.

Null values in all numeric columns are represented in SELECT output by a dot (period):

bigCol	intCol	doubleCol	vcCol	charCol	dateCol	timeCol	tsCol
.	.	.			.	.	.

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 18.

Table 2.2 *FedSQL Reserved Words A - D*

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

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 information for writing queries in the following ways.

caslibs identify the data sources that are available to a CAS session. caslibs are used to reference libraries in CAS, similarly to how librefs identify SAS libraries in SAS. To show existing data sources:

- From SAS, use the CASLIB _ALL_ LIST statement. See "CASLIB Statement" in *SAS Cloud Analytic Services: User's Guide*.
- When programming with actions, use the table.caslibInfo action. See "Tables Action Set" in *SAS Viya: System Programming Guide*.

caslibs also serve to organize in-memory tables. Typically, in-memory tables are loaded from the caslib's data source. To list tables that are available in memory:

- From SAS, use either of the following:
  - PROC CASUTIL with the LIST TABLES statement. See "LIST Statement" in *SAS Cloud Analytic Services: User's Guide*.
  - PROC DATASETS with a CAS LIBNAME engine libref.
- With action programming, use the table.tableInfo action.

To list information about the columns of in-memory tables:

- From SAS, use either of the following:
  - PROC CASUTIL with the CONTENTS statement. See "CONTENTS Statement" in *SAS Cloud Analytic Services: User's Guide*.
  - PROC CONTENTS with a CAS LIBNAME engine libref.
- With action programming, use the table.columnInfo action.

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 119. 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 124 and Totals from table "Sales" on page 125) 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 124, Totals from table "Sales" on page 125, and City from table "Customers" on page 120 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 124 and "Sales" on page 125, 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 124 and "Sales" on page 125 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 124 and column Totals from "Sales" on page 125. 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 124 and "Sales" on page 125. 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 38
- "Example of a Left Outer Qualified Join" on page 39
- "Example of Right Outer Qualified Join" on page 40
- "Example of a Full Outer Qualified Join" on page 41

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 124 and "Sales" on page 125. 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 38

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 124 and column Totals from table "Sales" on page 125. 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 38

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 124 and "Sales" on page 125. 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 38

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 125 and "Customers" on page 120. 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 38

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 value expressions and subqueries in the SELECT statements.

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.

Valid in: CAS

Syntax

expression [NOT] **BETWEEN** *expression* AND *expression*

Arguments

expression
specifies any valid SQL expression.

See "<sql-expression>" on page 71

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 71

CASE Expression

Selects result values that satisfy search conditions and value comparisons.

Valid in: CAS

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 71

"Overview of FedSQL Expressions and Subqueries" on page 45

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 71

result-expression

specifies an SQL expression that evaluates to a value.

See "<sql-expression>" on page 71

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 127

```
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 127

```
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 58
- "NULLIF Expression" on page 70
- <search-condition> in "SELECT Statement" on page 90

COALESCE Expression

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

Valid in: CAS

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 71

"Overview of FedSQL Expressions and Subqueries" on page 45

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 102.

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 54

DISTINCT Predicate

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

Valid in: CAS

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 71

"Overview of FedSQL Expressions and Subqueries" on page 45

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 92.

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 90

EXISTS Predicate

Tests whether a subquery returns one or more rows.

Valid in: CAS

Syntax

[NOT] **EXISTS** (*select-statement*)

Arguments

select-statement
specifies a subquery with the SELECT statement.

See "SELECT Statement" on page 90

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 90

IN Predicate

Tests set membership.

Valid in: CAS

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 71

"Overview of FedSQL Expressions and Subqueries" on page 45

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 127

```
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.

Valid in: CAS

Syntax

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

Arguments

expression
specifies any valid SQL expression.

See "<sql-expression>" on page 71

"Overview of FedSQL Expressions and Subqueries" on page 45

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 126

```
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 66
- "IS UNKNOWN Predicate" on page 67
- <search-condition> in the "SELECT Statement" on page 90

IS MISSING Predicate

Tests for a SAS missing value in a SAS native data store.

Valid in: CAS

Syntax

expression **IS** [NOT] **MISSING**

Arguments

expression
specifies any valid SQL expression.

See "<sql-expression>" on page 71

"Overview of FedSQL Expressions and Subqueries" on page 45

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 126

```
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 65
- <search-condition> in the "SELECT Statement" on page 90

IS NULL Predicate

Tests for a null value.

Valid in: CAS

Syntax

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

Arguments

expression
specifies any valid SQL expression.

See "<sql-expression>" on page 71

"Overview of FedSQL Expressions and Subqueries" on page 45

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 126

```
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 63
- <search-condition> in the "SELECT Statement" on page 90

IS TRUE Predicate

Tests for a true value.

Valid in: CAS

Syntax

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

Arguments

expression
specifies any valid SQL expression.

See "<sql-expression>" on page 71

"Overview of FedSQL Expressions and Subqueries" on page 45

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 126

```
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 62
- "IS UNKNOWN Predicate" on page 67

IS UNKNOWN Predicate

Tests for an unknown value.

Valid in: CAS

Syntax

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

Arguments

expression
specifies any valid SQL expression.

See "<sql-expression>" on page 71

"Overview of FedSQL Expressions and Subqueries" on page 45

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 62
- "IS TRUE Predicate" on page 66
- <search-condition> in the "SELECT Statement" on page 90

LIKE Predicate

Tests for a matching pattern.

Valid in: CAS

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 71

"Overview of FedSQL Expressions and Subqueries" on page 45

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. For information about the TRIM function, see *SAS FedSQL Language Reference*.

Example

Table: DENSITIES on page 122

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

NULLIF Expression

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

Valid in: CAS

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 19.

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 71

"Overview of FedSQL Expressions and Subqueries" on page 45

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 126

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

See Also

Expressions:

- "CASE Expression" on page 54
- "COALESCE Expression" on page 58

<sql-expression>

Produces a single value from a combination of symbols and operators or predicates.

Valid in: CAS

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 83

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 54 or "COALESCE Expression" on page 58.

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 53
- "DISTINCT Predicate" on page 59
- "EXISTS Predicate" on page 60
- "IN Predicate" on page 61
- "IS FALSE Predicate" on page 62.
- "IS MISSING Predicate" on page 63
- "IS NULL Predicate" on page 65
- "IS TRUE Predicate" on page 66
- "IS UNKNOWN Predicate" on page 67

- "LIKE Predicate" on page 68

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
1	CASE expression	See "CASE Expression" on page 54
2	**	raises to a power
	unary +, unary −	indicates a positive or negative number
3	*	multiplies
	/	divides
4	+	adds
	−	subtracts
5	||	concatenates

Group	Expressions, Operators, and Predicates	Description
6	[NOT] BETWEEN predicate	See “BETWEEN Predicate” on page 53.
	DISTINCT predicate	See “DISTINCT Predicate” on page 59
	[NOT] EXISTS predicate	See “EXISTS Predicate” on page 60
	[NOT] IN predicate	See “IN Predicate” on page 61
	IS [NOT] TRUE predicate	See “IS TRUE Predicate” on page 66
	IS [NOT] FALSE predicate	See “IS FALSE Predicate” on page 62
	IS [NOT] MISSING predicate	See “IS MISSING Predicate” on page 63
	IS [NOT] NULL predicate	See “IS NULL Predicate” on page 65
	IS [NOT] UNKNOWN predicate	See “IS UNKNOWN Predicate” on page 67
	LIKE predicate	See “LIKE Predicate” on page 68
7	=	equals
	^=, <>	does not equal
	>	is greater than
	<	is less than
	>=	is greater than or equal to
	<=	is less than or equal to
8	AND	indicates logical AND
9	OR	indicates logical OR
10	NOT	indicates logical NOT

SAS missing values and null values always appear as the smallest value in the collating sequence.

You can use parentheses to group values or to nest mathematical expressions. Parentheses make expressions easier to read and can also be used to change the order of evaluation of the operators. Evaluating expressions with parentheses begins at the deepest level of parentheses and moves outward. For example, SAS evaluates A+B*C as A+(B*C), although you can add parentheses to make it evaluate as (A+B)*C for a different result.

See Also

Statements:

- "SELECT Statement" on page 90
- "Overview of FedSQL Expressions and Subqueries" on page 45

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. For example, the DOLLARw.d format, which converts numeric values to a decimal monetary value, writes the numeric value `4503945867` as `$4,503,945,867.00`.

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. Formats are specified with the PUT function.

How to Format Output with the PUT Function

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

- The format can be applied to a string or a table column.
- You can apply both user-defined formats and formats that are provided by SAS.
- 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.
- FedSQL supports the same formats with the PUT function on the CAS server that it supports for third-party data sources in SAS 9.4. For a listing of formats, see Formats Supported with the PUT Function, by Category.

- 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.

See: "PUT Function" in *SAS FedSQL Language Reference*.

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.);

select put(x, best8.);
```

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. Use the CASFMTLIB= option to specify the location of your format library. Specify your CAS session name with the SESSREF= option.

```
cas mysess;

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

The resulting user-defined format can be applied to a CAS table as follows. 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 79. The output of the PUT function is shown in Figure 6.2 on page 79.

Figure 6.1 *Content of the Source EMPLOYEES Table*

EMP_NAME	HIRE_DATE	DEPT
Greg Welty	26NOV2004	20
Penny Jackson	26NOV2004	20
Edward Murray	26NOV2004	10
Ronald Thomas	26NOV2004	10

Figure 6.2 *Content of the Employees Table After the PUT Function Is Applied*

EMP_NAME	HIRE_DATE	DEPT
Greg Welty	26NOV2004	Research
Penny Jackson	26NOV2004	Research
Edward Murray	26NOV2004	Sales
Ronald Thomas	26NOV2004	Sales

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

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 National Language Support (NLS): Reference Guide*.

Category	Language Element	Description
Date and Time	NLDATE*w*.	Converts a SAS date value to the date value of the specified locale, and then writes the date value as a date.
	NLDATEMD*w*.	Converts the SAS date value to the date value of the specified locale, and then writes the value as the name of the month and the day of the month.
	NLDATEMN*w*.	Converts a SAS date value to the date value of the specified locale, and then writes the value as the name of the month.
	NLDATEW*w*.	Converts a SAS date value to the date value of the specified locale, and then writes the value as the date and the day of the week.
	NLDATEWN*w*.	Converts the SAS date value to the date value of the specified locale, and then writes the date value as the day of the week.
	NLDATEYM*w*.	Converts the SAS date value to the date value of the specified locale, and then writes the date value as the year and the name of the month.
	NLDATEYQ*w*.	Converts the SAS date value to the date value of the specified locale, and then writes the date value as the year and the quarter.
	NLDATEYR*w*.	Converts the SAS date value to the date value of the specified locale, and then writes the date value as the year.
	NLDATEYW*w*.	Converts the SAS date value to the date value of the specified locale, and then writes the date value as the year and the week.
	NLDATMAP*w*.	Converts a SAS datetime value to the datetime value of the specified locale, and then writes the value as a datetime with a.m. or p.m.
	NLDATMDT*w*.	Converts the SAS datetime value to the datetime value of the specified locale, and then writes the value as the name of the month, day of the month, and year.
	NLDATMMD*w*.	Converts the SAS datetime value to the datetime value of the specified locale, and then writes the value as the name of the month and the day of the month.
	NLDATMMN*w*.	Converts the SAS datetime value to the datetime value of the specified locale, and then writes the value as the name of the month.

Category	Language Element	Description
	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.
	NLNUM*w.d*	Writes the numeric format of the local expression in the specified locale.

Category	Language Element	Description
	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 Reference

FedSQL for CAS supports the same formats as FedSQL for SAS 9.4. See reference information about the formats supported with FedSQL in Formats Supported with the PUT Function, by Category in *SAS FedSQL Language Reference*.

Chapter 7

FedSQL Functions

Overview of FedSQL Functions in CAS

A FedSQL function performs a computation on FedSQL expressions and returns either a single value or a set of values if the FedSQL function is an aggregate function. In other SQL environments, aggregate functions are also known as *set functions*. Most other functions use arguments supplied by the user, but a few obtain their arguments from the operating environment.

FedSQL for CAS supports the same functions that are provided for FedSQL in SAS 9.4, with the following exceptions:

- The CAST function is not supported in CAS.
- FedSQL for CAS does not support use of DS2 packages in expressions.

When using FedSQL functions, note these points:

- Within the functions, the FedSQL expressions in function arguments are limited to the SQL expressions that are supported in CAS. For more information, see “<sql-expression>” on page 71.
- The FedSQL language supports more data types than are used in CAS tables. When the data types of the arguments in the function expression are not supported in a CAS table, FedSQL performs a type conversion on the arguments so that the arguments have the appropriate data type. For CAS, columns of all FedSQL numeric types are converted to DOUBLE. The functions operate on CHAR and VARCHAR columns as documented.

For information about FedSQL functions, see FedSQL Functions by Category.

Integration with DS2

Currently, FedSQL functions can be used only in the SET statement of a DS2 program that runs in CAS.

Specifying Function Arguments in FedSql.execDirect

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 the conventions for these data types. 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 INTNX function is an example of a function that takes FedSQL constants. In the following example, the INTNX function specifies the constants YEAR and SAME and a date value.

```
select put(intnx('year', date'2011-03-15', 5, 'same'), date9.);
```

The SCAN function is an example of a function that takes character strings:

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

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 put(intnx(''year'', date''2011-03-15'',
 5, ''same''), date9.)';
quit;
```

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

Understanding Function Output

FedSQL Date and Time Functions

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 use the PUT function to apply a SAS format to the value. 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;
```

The PUT function applies the SAS DATEw. format to the function request. For information about this format, see "DATEw. Format" in *SAS FedSQL Language Reference*.

For information about the PUT function, see "PUT Function" in *SAS FedSQL Language Reference*.

The Output Delivery System and FedSQL

The interface that you use to submit a FedSQL request can affect the length of numeric values displayed for a FedSQL function. For example, PROC FEDSQL displays 8 characters for numeric functions, but the fedSql.execDirect action displays 12 characters for numeric functions. To display numeric output with the full precision of which FedSQL is capable, use the PUT function with the BEST16. format with the FedSQL functions. The following example shows how to format a FedSQL BETA function request with the PUT function:

```
select PUT (beta(5,3), best16.) as Beta;
```

This statement returns the output 0.00952380952381. For more information about the format, see "BESTw. Format" in *SAS FedSQL Language Reference*.

Functions Reference

FedSQL for CAS supports the same functions that are provided for FedSQL in SAS 9.4. See reference information for FedSQL functions in FedSQL Functions by Category in *SAS FedSQL Language Reference*.

Chapter 8

FedSQL Statements

Dictionary

CREATE TABLE Statement

Creates a new table from one or more existing tables.

Valid in: CAS

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 111

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 88

"Overview of FedSQL Expressions and Subqueries" on page 45

"SELECT Statement" on page 90

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 89

Table Options

- "LABEL= Table Option" on page 112
- "REPLACE= Table Option" on page 114

DROP TABLE Statement

Removes an in-memory table from the CAS session.

Valid in: CAS

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 87

SELECT Statement

Retrieves columns and rows of data from tables.

Valid in: CAS

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 92
- "FROM Clause" on page 94
- "WHERE Clause" on page 103
- "GROUP BY Clause" on page 103
- "HAVING Clause" on page 104
- "ORDER BY Clause" on page 105

- "LIMIT Clause" on page 106
- "OFFSET Clause" on page 107

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 46

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

See "<sql-expression>" on page 71

table*.
selects all columns in the table.

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

See "Table Aliases" on page 95

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 and reserved words can be used if they are quoted. You must use double quotation marks. See "Delimited Identifiers" on page 18. 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 98

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 98

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 102

(<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 46

table
specifies the name of a table. The name can be in the following forms:

- *table-name*
- *caslib.table-name*

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 107

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

See "ON and USING Clauses" on page 101

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 27.

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 96. 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 119.

```
/* 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 98.

```
/* 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.

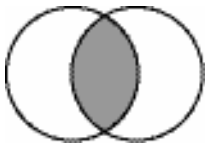
Qualified Joins

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 99.

```
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;
```

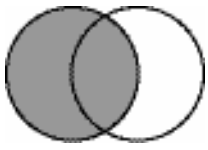
Outer Joins

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;
```

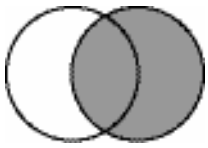
Output 8.4 *Left Outer Join - GrainProducts and Sales Table*

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;
```

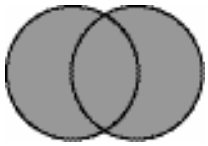
Output 8.5 *Right Outer Join - GrainProducts and Sales Table*

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 107

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 71

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 107

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]]

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 71

ASC
orders the data in ascending order. This is the default order.

DESC
orders the data in descending order.

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 8.2 *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 8.3 *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 8.4 *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 71

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 table 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.

How to Specify FedSQL Statement Table Options

Specify a FedSQL statement table option immediately after the table name, within braces (that is, { }) and including the keyword OPTIONS. To specify several table options, separate them with spaces or commas. For example:

```
create table newtable {options replace=true copies=3} as select * from casdblib.table;
```

CAUTION:

You cannot have a space between the left brace { and the OPTIONS keyword. A space results in a syntax error.

Dictionary

COMPRESS= Table Option

Specifies whether rows are compressed in a new output CAS table.

Valid in: CAS

Category: Table Control

Default: FALSE

Syntax

COMPRESS=[TRUE | FALSE]

Optional Arguments

TRUE
specifies that the rows in the newly created CAS table are compressed.

FALSE
specifies that the rows in the newly created table are not compressed.

Details

Compressing a table is a process that reduces the number of bytes required to represent each row. Advantages of compressing a table include reduced storage requirements for the table and fewer I/O operations necessary to read or write to the data during processing. However, more CPU resources are required to read a compressed table (because of the overhead of uncompressing each row). Also, there are situations where the resulting file size might increase rather than decrease.

After a table is compressed, the setting is a permanent attribute of the table. To change the setting, you must re-create the table.

LABEL= Table Option

Specifies a label for an output table.

Valid in: CAS

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.

Valid in: CAS

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.

REPLICATION= Table Option

specifies the number of copies of the table to make for fault tolerance.

Valid in: CAS

Category: Table Control

Alias: COPIES=

Default: 1

Syntax

REPLICATION= *number*

Arguments

number

specifies the number of copies of the table to make for fault tolerance. Larger values result in slower performance and use more memory, but provide high availability for data in the event of a node failure. The minimum value is 0.

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

Word1	Word2
every	THING
no	BODY

Customers

Code:

The following DS2 statements can be used to create table Customers in CAS. Submit the DS2 statements in the ds2.runDS2 action or in PROC DS2 with the SESSREF= option.

```
data customers;
  dcl int Custid;
  dcl varchar(16) Name;
  dcl varchar(64) Address;
  dcl varchar(16) City;
  dcl char(2) State;
  dcl varchar(16) Country;
  dcl char(16) Phone;
  dcl date InitOrder;
  method run();
   Custid=1; Name='Peter Frank'; Address='300 Rock Lane';  City='Boulder';
State='CO'; Country='United States'; Phone='3039564321';
InitOrder=date '2012-01-14';
output;
   Custid=2; Name='Jim Stewart'; Address='1500 Lapis Lane';  City='Little Rock';
State='AR'; Country='United States'; Phone='8705553978'; InitOrder=date '2012-03-20';
output;
   Custid=3; Name='Janet Chien'; Address='75 Jujitsu';  City='Nagasaki';
State=' '; Country='Japan'; Phone='01181956879932'; InitOrder=date '2012-06-07';
output;
   Custid=4; Name='Qing Ziao'; Address='10111 Karaje';  City='Tokyo';
State=' '; Country='Japan'; Phone='0118136774351'; InitOrder=date '2012-10-12';
output;
   Custid=5; Name='Humberto Sertu'; Address='876 Avenida Blanca'; City='Buenos Aires';
State=' '; Country='Argentina'; Phone='01154118435029'; InitOrder=date '2012-12-15';
output;
 end;
enddata;
run;
```

Content:

Table A1.2 *Customers*

Custid	Name	Address	City	State	Country	Phone	InitOrder
1	Peter Frank	300 Rock Lane	Boulder	CO	United States	3039564321	14JAN2012
2	Jim Stewart	1500 Lapis Lane	Little Rock	AR	United States	8705553978	20MAR2012
3	Janet Chien	75 Jujitsu	Nagasaki		Japan	01181956879932	07JUN2012
4	Qing Ziao	10111 Karaje	Tokyo		Japan	0118136774351	12OCT2012
5	Humberto Sertu	876 Avenida Blanca	Buenos Aires		Argentina	01154118435029	15DEC2012

CustonLine

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
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

Code:

The following DS2 statements can be used to create table Employees in CAS. Submit the DS2 statements in the ds2.runDS2 action or in PROC DS2 with the SESSREF= option.

```
data employees;
  dcl int EmpId;
  dcl int Dept;
  dcl varchar(30) Emp_Name;
  dcl varchar(50) Pos;
  dcl date Hire_Date;
  method run();
    EmpId=1; Dept=10; Emp_Name='Jim Barnes'; Pos='Manager';
Hire_Date=date '2000-11-26'; output;
    EmpId=2; Dept=20; Emp_Name='Clifford James'; Pos='Manager';
Hire_Date=date '2000-11-26'; output;
    EmpId=3; Dept=30; Emp_Name='Barbara Sandman'; Pos='Manager';
```

```
Hire_Date=date '2000-11-26'; output;
    EmpId=4; Dept=40; Emp_Name='William Baylor'; Pos='Manager';
Hire_Date=date '2000-11-26'; output;
    EmpId=5; Dept=20; Emp_Name='Greg Welty'; Pos='Developer';
Hire_Date=date '2004-11-26'; output;
    EmpId=6; Dept=20; Emp_Name='Penny Jackson'; Pos='Developer';
Hire_Date=date '2004-11-26'; output;
    EmpId=7; Dept=10; Emp_Name='Edward Murray'; Pos='Sales Associate';
Hire_Date=date '2004-11-26'; output;
    EmpId=8; Dept=20; Emp_Name='Ronald Thomas'; Pos='Sales Associate';
Hire_Date=date '2004-11-26'; output;
    EmpId=9; Dept=30; Emp_Name='Elsie Marks'; Pos='Executive Assistant';
Hire_Date=date '2005-02-11'; output;
    EmpId=10; Dept=40; Emp_Name='Bruno Kramer'; Pos='Grounds support technician';
Hire_Date=date '2005-11-02'; output;
  end;
 enddata;
run;
```

Content:

EmpID	Dept	Emp_Name	Pos	Hire_Date
1	10	Jim Barnes	Manager	26NOV2000
2	20	Clifford James	Manager	26NOV2000
3	30	Barbara Sandman	Manager	26NOV2000
4	40	William Baylor	Manager	26NOV2000
5	20	Greg Welty	Developer	26NOV2004
6	20	Penny Jackson	Developer	26NOV2004
7	10	Edward Murray	Sales Associate	26NOV2004
8	10	Ronald Thomas	Sales Associate	26NOV2004
9	30	Elsie Marks	Executive Assistant	11FEB2005
10	40	Bruno Kramer	Grounds support technician	02NOV2005

GrainProducts

Table A1.5 GrainProducts

Prodid	Product
1424	Rice
3421	Corn
3234	Wheat
3485	Oat

Products

Code:

The following DS2 statements can be used to create table Products in CAS. Submit the DS2 statements in the ds2.runDS2 action or in PROC DS2 with the SESSREF= option.

```
data products;
  dcl int ProdId;
  dcl char(10) Product;
  method run();
    ProdId=3234; Product='Rice'; output;
    ProdId=1424; Product='Corn'; output;
    ProdId=3421; Product='Wheat'; output;
    ProdId=3421; Product='Oat'; output;
    ProdId=3975; Product='Barley'; output;
  end;
 enddata;
run;
```

Content:

Table A1.6 Products

Prodid	Product
3234	Rice
1424	Corn
3421	Wheat

Prodid	Product
3422	Oat
3975	Barley

Sales

Code:

The following DS2 statements can be used to create table Sales in CAS. Submit the DS2 statements in the ds2.runDS2 action or in PROC DS2 with the SESSREF= option.

```
data sales;
  dcl int ProdId;
  dcl int CustId;
  dcl bigint Totals;
  dcl varchar(32) Country;
  method run();
    ProdId=3234; CustId=1; Totals=189400;  Country='United States'; output;
    ProdId=1424; CustId=3; Totals=555789;  Country='Japan';         output;
    ProdId=3421; CustId=4; Totals=781183;  Country='Japan';         output;
    ProdId=3421; CustId=2; Totals=2789654; Country='United States'; output;
    ProdId=3975; CustId=5; Totals=899453;  Country='Argentina';     output;
  end;
 enddata;
run;
```

Content:

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

Code:

The following DS2 statements can be used to create table WorldCityCoords in CAS. Submit the DS2 statements in the ds2.runDS2 action or in PROC DS2 with the SESSREF= option.

```
data worldcitycoords;
  dcl varchar(16) City;
  dcl varchar(16) Country;
  dcl double Latitude;
  dcl double Longitude;
  method run();
    City='Algiers'; Country='Algeria'; Latitude=37; Longitude=3; output;
    City='Amsterdam'; Country='Netherlands'; Latitude=52; Longitude=5; output;
    City='Beijing'; Country='China'; Latitude=40; Longitude=116; output;
    City='Bombay'; Country='India'; Latitude=19; Longitude=73; output;
    City='Calcutta'; Country='India'; Latitude=22; Longitude=88; output;
    City='Caracas'; Country='Venezuela'; Latitude=10; Longitude=-67; output;
    City='Geneva'; Country='Switzerland'; Latitude=46; Longitude=6; output;
    City='Hong Kong'; Country='China'; Latitude=22; Longitude=114; output;
    City='Lagos'; Country='Nigeria'; Latitude=6; Longitude=3; output;
    City='Madrid'; Country='Spain'; Latitude=40; Longitude=-3; output;
    City='Shangai'; Country='China'; Latitude=31; Longitude=121; output;
    City='Zurich'; Country='Switzerland'; Latitude=47; Longitude=8; output;
  end;
 enddata;
run;
```

Content:

Table A1.8 *WorldCityCoords*

City	Country	Latitude	Longitude
Algiers	Algeria	37.000000	3.000000
Amsterdam	Netherlands	52.000000	5.000000
Beijing	China	40.000000	116.000000
Bombay	India	19.000000	73.000000
Calcutta	India	22.000000	88.000000
Caracas	Venezuela	10.000000	-67.000000
Geneva	Switzerland	46.000000	6.000000

City	Country	Latitude	Longitude
Hong Kong	China	22.000000	114.000000
Lagos	Nigeria	6.000000	3.000000
Madrid	Spain	40.000000	-3.000000
Shangai	China	31.000000	121.000000
Zurich	Switzerland	47.000000	8.000000

WorldTemps

Code:

The following DS2 statements can be used to create table WorldTemps in CAS. Submit the DS2 statements in the ds2.runDS2 action or in PROC DS2 with the SESSREF= option.

```
data worldtemps;
  dcl varchar(16) City;
  dcl varchar(16) Country;
  dcl double AvgHigh;
  dcl double AvgLow;
  method run();
    City='Algiers'; Country='Algeria'; AvgHigh=90; AvgLow=45; output;
    City='Amsterdam'; Country='Netherlands'; AvgHigh=79; AvgLow=33; output;
    City='Beijing'; Country='China'; AvgHigh=86; AvgLow=17; output;
    City='Bombay'; Country='India'; AvgHigh=90; AvgLow=68; output;
    City='Calcutta'; Country='India'; AvgHigh=97; AvgLow=56; output;
    City='Caracas'; Country='Venezuela'; AvgHigh=83; AvgLow=57; output;
    City='Geneva'; Country='Switzerland'; AvgHigh=76; AvgLow=28; output;
    City='Hong Kong'; Country='China'; AvgHigh=89; AvgLow=51; output;
    City='Lagos'; Country='Nigeria'; AvgHigh=90; AvgLow=75; output;
    City='Madrid'; Country='Spain'; AvgHigh=89; AvgLow=36; output;
    City='Shangai'; Country='China'; AvgHigh=.; AvgLow=33; output;
    City='Zurich'; Country='Switzerland'; AvgHigh=78; AvgLow=25; output;
    end;
 enddata;
run;
```

Content:

Table A1.9 *WorldTemps*

City	Country	AvgHigh	AvgLow
Algiers	Algeria	90.000000	45.000000
Amsterdam	Netherlands	79.000000	33.000000
Beijing	China	86.000000	17.000000
Bombay	India	90.000000	68.000000
Calcutta	India	97.000000	56.000000
Caracas	Venezuela	83.000000	57.000000
Geneva	Switzerland	76.000000	28.000000
Hong Kong	China	89.000000	51.000000
Lagos	Nigeria	90.000000	75.000000
Madrid	Spain	89.000000	36.000000
Shanghai	China		33.000000
Zurich	Switzerland	78.000000	25.000000

Appendix 2

ICU License Agreement

Recommended Reading

- *SAS Cloud Analytic Services: Fundamentals*
- *SAS Cloud Analytic Services: CASL Reference*
- *Getting Started with SAS Viya for Python*
- *SAS Viya: System Programming Guide*
- *SAS Viya Administration: Cloud Analytic Services Authorization*
- *Base SAS 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

V