SAS® Viya® 3.5: FedSQL Programming for SAS® Cloud Analytic Services
## Contents

*What's New* .................................................................................................................. vii

### PART 1 Concepts 1

<table>
<thead>
<tr>
<th>Chapter 1 / Introduction to SAS FedSQL for CAS</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction to the SAS FedSQL Language in CAS</td>
<td>3</td>
</tr>
<tr>
<td>Running FedSQL Programs in CAS</td>
<td>3</td>
</tr>
<tr>
<td>Supported Statements</td>
<td>4</td>
</tr>
<tr>
<td>Supported Data Sources</td>
<td>5</td>
</tr>
<tr>
<td>FedSQL Implicit Pass-Through Facility in CAS</td>
<td>5</td>
</tr>
<tr>
<td>FedSQL Explicit Pass-Through Facility</td>
<td>7</td>
</tr>
<tr>
<td>FedSQL Federated Queries</td>
<td>8</td>
</tr>
<tr>
<td>FedSQL Output Tables</td>
<td>8</td>
</tr>
<tr>
<td>Intended Audience</td>
<td>9</td>
</tr>
<tr>
<td>Syntax Conventions for the FedSQL Language</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 2 / How FedSQL Runs in CAS</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>How FedSQL Runs in CAS</td>
<td>11</td>
</tr>
<tr>
<td>Viewing the FedSQL Query Plan</td>
<td>14</td>
</tr>
<tr>
<td>FedSQL Query Walk-Through</td>
<td>16</td>
</tr>
<tr>
<td>Optimizing FedSQL Performance by Modifying the FedSQL Query Plan</td>
<td>22</td>
</tr>
<tr>
<td>Using Cntl Instructions That Affect FedSQL Implicit Pass-Through</td>
<td>24</td>
</tr>
<tr>
<td>Using the Dynamic Cardinality Instruction</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 3 / Joining Data with FedSQL</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview of Joins</td>
<td>28</td>
</tr>
<tr>
<td>Example: Typical Two-Table Join</td>
<td>30</td>
</tr>
<tr>
<td>Example: Typical Three-Table Join</td>
<td>31</td>
</tr>
<tr>
<td>Example: Simple Join Including All Columns</td>
<td>32</td>
</tr>
<tr>
<td>Example: Equijoin Including All Columns</td>
<td>33</td>
</tr>
<tr>
<td>Example: Simple Cross Join</td>
<td>34</td>
</tr>
<tr>
<td>Example: Cross Join with Specified Columns and a WHERE Clause</td>
<td>36</td>
</tr>
<tr>
<td>Example: Qualified Join with an ON Clause</td>
<td>37</td>
</tr>
<tr>
<td>Example: Qualified Join with a USING Clause</td>
<td>38</td>
</tr>
<tr>
<td>Example: Qualified Join with an ON Clause and a WHERE Clause</td>
<td>40</td>
</tr>
<tr>
<td>Example: Natural Join</td>
<td>41</td>
</tr>
<tr>
<td>Example: Natural Join with a WHERE Clause</td>
<td>42</td>
</tr>
<tr>
<td>Understanding Inner and Outer Join Types</td>
<td>43</td>
</tr>
<tr>
<td>Example: Inner Join</td>
<td>43</td>
</tr>
<tr>
<td>Example: Left Outer Qualified Join</td>
<td>44</td>
</tr>
<tr>
<td>Example: Left Outer Natural Join</td>
<td>45</td>
</tr>
<tr>
<td>Example: Right Outer Qualified Join</td>
<td>46</td>
</tr>
<tr>
<td>Example: Right Outer Natural Join</td>
<td>47</td>
</tr>
<tr>
<td>Example: Full Outer Qualified Join</td>
<td>48</td>
</tr>
</tbody>
</table>
Appendix 1 / Trouble Shooting: FedSQL Error Codes
Resolving FedSQL Errors

Appendix 2 / Tables Used in Examples
Overview of Sample Tables
AfewWords
Customers
CustomLine
Densities
Employees
Numbers1
Numbers2
Numbers3
Products
Sales
WorldCityCoords
WorldTemps

Appendix 3 / ICU License Agreement
ICU License - ICU 1.8.1 and later
What’s New

What’s New in FedSQL Programming for CAS

Overview

FedSQL for CAS has the following new functionality in SAS Viya 3.5:

- three new instructions for the Cntl option
- a new showStages option
- support for the CAST function in CAS
- support for the UNION set operator in CAS
- support for the VARBINARY data type in CAS tables.

Here is a list of additional changes:

- The output of the METHOD option has changed.
- The behavior of the DROP TABLE statement and REPLACE table option has changed.
- There are performance improvements.
- Trouble-shooting error codes have been added.
- The documentation has been enhanced.

New Cntl Instructions

Some instructions have been added that enable you to influence the execution of the FedSQL SQL plan.

- An optional dynamicCardinality instruction for the Cntl option directs the FedSQL query planner to perform a cardinality estimation before creating a query plan.
- An optional optimizeVarbinaryPrecision instruction for the Cntl option optimizes VARBINARY precision by using a precision that is appropriate to the actual data, instead of the declared precision for VARBINARY columns.
- An optional optimizeVarcharPrecision instruction for the Cntl option optimizes VARCHAR precision by using a precision that is appropriate to the actual data, instead of the declared precision for VARCHAR columns.

For more information about the Cntl instructions, see "Optimizing FedSQL Performance by Modifying the FedSQL Query Plan" on page 22.

The optimizeVarbinaryPrecision and optimizeVarcharPrecision instructions can also be set as environment variables. For more information, see SAS Viya Administration: SAS Cloud Analytic Services

showStages Option

The new showStages option returns FedSQL query execution details, including times for intermediate stages, in addition to the FedSQL query plan. “Viewing the FedSQL Query Plan” on page 14 compares Method and showStages output.

CAST Function

The CAST function converts a column from one data type to another and also enables you to change the column’s length. For information about the CAST function, see “CAST Function” in SAS FedSQL Language Reference. See “Understanding Function Output” on page 109 for information about one use of the CAST function.

UNION Set Operator

The UNION set operator combines the result sets of two SELECT queries and returns unique rows from both result sets. Columns are combined by position by default. An optional CORRESPONDING keyword enables you to combine columns based on column name instead. An optional BY keyword enables you to submit a column list to the CORRESPONDING keyword. For more information, see “UNION Operator” on page 127 and “Query Expressions and Subqueries” on page 51.

DROP TABLE Statement and REPLACE= Table Option

In earlier releases of SAS Viya, the DROP TABLE statement and REPLACE= table option considered global tables when enforcing name uniqueness rules and would drop and replace a table that was previously promoted. Beginning in SAS Viya 3.5, the DROP TABLE statement and REPLACE= table option operate exclusively on CAS session tables. For more information, see “DROP TABLE Statement” on page 115 and “REPLACE= Table Option” on page 134.
Performance Improvements

See “Processing Differences between SAS Viya 3.5 and Earlier Releases” on page 12.

Trouble-Shooting Error Codes

Some FedSQL error messages that are written to the SAS log now contain an internal error code. If you contact SAS Technical Support, supply these codes, which are hexadecimal numbers that can help SAS more quickly determine the source of the problem. For more information, see Appendix 1, “Trouble Shooting: FedSQL Error Codes,” on page 139.

Documentation Enhancements

The following existing topics have been updated:

The following topics are new:
- “Aggregate Processing When GROUP BY Is Not Used” on page 13
- “Handling of Date, Time, and Datetime Values in CAS” on page 14
- “Using Cntl Instructions That Affect FedSQL Implicit Pass-Through” on page 24
- “Using the Dynamic Cardinality Instruction” on page 25
What’s New
PART 1

Concepts

Chapter 1

Introduction to SAS FedSQL for CAS

3

Chapter 2

How FedSQL Runs in CAS

11

Chapter 3

Joining Data with FedSQL

27

Chapter 4

FedSQL Expressions and Subqueries

51
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 and create new CAS tables from existing tables. 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 in CAS libraries. The FedSQL language
  supports a more expanded syntax for SAS libraries than it does for CAS
  libraries. You must use the FedSQL syntax described in this book for requests
  submitted to the CAS server. For information about how to submit FedSQL
  statements to a CAS library with PROC FEDSQL, 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 the fedSql.execDirect action 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
  information about how to submit FedSQL statements with 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 "Use the
Native Language to Operate on a FedSQL Result Set" 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 “FedSQL Statements” on page 113.

Supported Data Sources


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 before processing with an execDirect action.

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

A serial data connector can use one or more CAS grid nodes to connect to and read data from a DBMS. The number of CAS grid nodes that are used is controlled with the numReadNodes= parameter of the table.addCaslib action. Use of execDirect with a serial data connector against an unloaded CAS table is always performed with numReadNodes=1, which means the request is processed on the controller.

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 the full query cannot be passed to the data source, the request is processed locally on the CAS server.

FedSQL supports implicit SQL pass-through for the following data sources through SAS data connectors:

- Amazon Redshift
- Apache Spark (new in SAS Viya 3.4)
- DB2 (UNIX)
- Hadoop (Hive)
- Impala
- databases that are compliant with JDBC (new in SAS Viya 3.4)
- databases that are compliant with ODBC
- Oracle
- PostgreSQL
- SAP Hana
- Teradata (UNIX)

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 FedSQL Implicit Pass-Through

Here are the requirements for FedSQL implicit pass-through in CAS:

- 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 unloaded tables that exist in different caslibs are automatically loaded into CAS for processing.
- The query cannot contain an ORDER BY clause.
- None of the tables in the SQL query can have CAS row-level or column-level security where the number of columns returned for the table is less than the number of columns that actually exist in the table.

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.

---

**FedSQL Explicit Pass-Through Facility**

**Overview**

Beginning with SAS Viya 3.4, the FedSQL explicit pass-through facility enables you to connect to a data source and send SQL statements directly to that data source for execution. This facility also enables you to use the syntax of your data source, regardless of whether it meets ANSI standards for SQL.

FedSQL in CAS supports explicit SQL pass-through through the use of a CONNECTION TO component in the SELECT statement’s FROM clause. The CONNECTION TO component enables you to submit native SQL requests that produce a result set.

**How to Use the FedSQL Explicit Pass-Through Facility**

The CONNECTION TO component of the SELECT statement FROM clause has the following syntax:

```sql
FROM CONNECTION TO caslib (native-syntax) [[AS] alias]
```

- `caslib` specifies the name of a caslib in the existing CAS session.
- `native-syntax` specifies a SELECT-type query (not DDL) to be run on the caslib's driver.
**alias**

provides a name for the result set that is produced by the native query.

Example:

```sql
select oo.i, oo.rank, ff.onoff
from connection to caslib1
    ( select i, rank() over (order by j) rank from table_a ) oo,
connection to caslib2
    ( select distinct i, iif(k > 0.5, 1, 0) as onoff from table_a ) ff
where oo.i = ff.i order by 1;
```

For more information, see “SELECT Statement” on page 116.

---

**Conditions for Explicit Pass-Through**

The native syntax must be valid for the data source. The statements that you use must produce a result set.

---

**FedSQL Federated Queries**

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 caslibs must have been assigned previously and reference a SAS data connector. The tables from the specified caslibs are then loaded into CAS for processing.

Here is an example of a federated query:

```sql
select ora.city, ora.state, ora.zip
from Oracle.table ora, mycas.table mycas, Teradata.table tera
where ora.zip = mycas.zip and mycas.zip = tera.zip;
```

---

**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.
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 the data.
Nodes in the FedSQL Query Plan

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

- Aggregate
- Append (UNION)
- Group
- HashJoin
- Limit
- MergeJoin
- NestLoop
- Result
- SeqScan (when it is the root of the plan)
- Sort (when ORDER BY is used)
- SubqueryScan
- Unique

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

- Append (UNION)
- Group
- HashJoin
- MergeJoin
- NestLoop
- SeqScan
- Unique

In summary, Reads and Joins are processed in parallel, except some full outer joins. FedSQL for CAS provides several ways to view the query plan for any given FedSQL request, including a new option that returns query execution details. For more information, see “Viewing the FedSQL Query Plan” on page 14.

Processing Differences between SAS Viya 3.5 and Earlier Releases

In previous releases of SAS Viya, the fedSql.execDirect action serialized many operations on one worker.

Beginning with SAS Viya 3.5, the action partitions operations that include computed columns by creating temporary intermediate tables. The following operations, which were previously serialized on a one worker, can now be performed on multiple workers:
Full outer joins, except for the following:

- Full outer joins where the join condition is an inequality condition ($t1.x <> t2.x$)
- Full outer joins that have an equality condition in which one side of the equation contains column references from more than one table ($t1.x + t2.y = ?$).
- GROUP BY aggregations where one or more group expressions are computed columns
- Joins on columns that use different SAS formats.
- SELECT DISTINCT operations with computed columns.

LIMIT and OFFSET operations are still serialized on one worker.

FedSQL Sorting and Grouping in CAS

In SAS Viya releases before SAS Viya 3.5, the execDirect action sorted locally any partitions that were returned from a partitioned open. The sorting was performed in RAM, with no backing store. If an operation's memory needs exceeded RAM, the operation failed. Beginning with SAS Viya 3.5, the execDirect action enables the CAS server to sort partitions. The CAS server has a disk cache backing store, which helps avoid memory failures.

As a result of the update, you will see a change in how empty and blank values in VARCHAR columns are grouped in GROUP BY and SELECT DISTINCT operations.

GROUP BY:

In earlier SAS Viya releases, FedSQL placed empty VARCHAR values in one group and VARCHAR values of all blanks in a different group. Beginning with SAS Viya 3.5, the values are placed in the same group.

SELECT DISTINCT:

In earlier SAS Viya releases: For the purpose of eliminating duplicate rows, FedSQL treated empty VARCHAR values and VARCHAR values of all blanks as unequal. Beginning with SAS Viya 3.5, the values are considered equal.

FedSQL processes ORDER BY queries locally in a single thread on one CAS worker, without a backing store for the sort output.

Aggregate Processing When GROUP BY Is Not Used

When the GROUP BY statement is not specified in an aggregate query, FedSQL submits some of the aggregate queries to the CAS aggregate action from the aggregation action set. It processes other aggregation queries itself. When the aggregate action processes the query, the data is processed in parallel and does not require an auto-partition of the input table. When FedSQL processes the query, the aggregation is performed in one thread on one worker. This is not new to SAS Viya 3.5.

These conditions prevent FedSQL from passing aggregate queries to the aggregate action:
n The aggregation query contains a GROUP BY clause.

n The argument to an aggregate function is not a simple column reference. These are examples of queries that are not sent to the aggregate action: \( \max(a+b) \) and \( \min(\text{abs}(c)) \).

n The argument to an aggregate function has an integer data type. That is, in \( \max(a) \), the CAS data type of column \( a \) is int32 or int64.

n The SELECT list contains computed columns other than aggregate functions. This is an example of computed columns that are not passed: \( \max(a) + \max(b) \).

The performance of the COUNT function is improved, beginning with SAS Viya 3.5. When COUNT(*) is specified with no other select list elements and no GROUP BY (for example, select count(*) from t), FedSQL performs the count by reading CAS table metadata. Any additional syntax disables the optimization.

**Handling of Date, Time, and Datetime Values in CAS**

FedSQL treats date, time, and datetime values as ANSI standard DATE, TIME, and TIMESTAMP types. For data sources that do not support DATE, TIME, and TIMESTAMP values, like the CAS server, FedSQL converts the types to a DOUBLE and applies a SAS format to them. This is useful when reading data from a DBMS into CAS, but less useful for creating a new CAS table from an existing CAS table.

Beginning with SAS Viya 3.5, when FedSQL creates a CAS table from an existing CAS table, it propagates the existing date, time, and datetime formats from the source columns.

To understand the effect of the change, consider the following code:

```sas
data mycas.test;
attrib dt format= datetime16.;
dt=datetime();
run;

proc fedsql sessref=mysess;
create table test2 as select * from mycas.test;
quit;
```

Before SAS Viya 3.5, FedSQL for CAS produced a column in the test2 table with a DATETIME25.6 format. Beginning with SAS Viya 3.5, the column in the new test2 table has the same format (DATETIME16.) that it had in the source table.

**Viewing the FedSQL Query Plan**

**Options for Viewing the FedSQL Query Plan**

To see the query plan for a given FedSQL request, set one of the following options:
Method
prints a brief text description of the nodes and stages in the query plan for a
given request and writes the output to the client log.

Information about the query plan is gathered when the query is executed, unless
you specify the validateOnly option with the Method option. When you specify
validateOnly with Method, FedSQL prints the query plan without executing the
query. Because the query is not executed, information about query stages is
omitted from the query plan. For an example of Method output with and without
the validateOnly option, see “FedSQL Query Walk-Through” on page 16.

Note: The output of the Method option has changed in SAS Viya 3.5. The
Method option no longer returns the stage query and the number of SQL threads
used by the query plan.

showStages
New in SAS Viya 3.5, this option prints the information returned by the Method
option. In addition, it prints the stage query, the number of SQL threads used by
each stage, and the following execution details:
- Time for each intermediate stage
- Number of output rows from each intermediate stage
- Elapsed time for each stage and for the entire action
- Whether a table was replicated to all workers, or auto-partitioned

The information is gathered when the query is executed and is written to the
client log. Do not specify validateOnly with showStages. Execution details cannot
be printed if the query is not executed. See “showStages Output” on page 18
for an example of the output produced by showStages.

How to Specify the Query Plan Options

PROC FEDSQL

In PROC FEDSQL, Method and NOEXEC are specified as procedure options. The
keyword Method is preceded by an underscore (_METHOD).

Here are examples of how the procedure options are specified.

To print the query plan only:

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

To print the query plan while executing the query:

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

showStages is a Cntl instruction. Here is an example of how showStages is
specified in PROC FEDSQL:

```sas
proc fedsql sessref=mysess cntl=(showStages);
```
FedSQL statements;
quit;

Note that showStages is specified within parentheses.

fedSql.execDirect

In the fedSql.execDirect action, Method, validateOnly, and showStages are specified as fedSql.execDirect action parameters. Here are examples of how the parameters are specified.

To print the query plan only:

```plaintext
test
fedsql.execdirect
  method=true
  validateOnly=true
  query="...FedSQL statements...";
quit;
```

To print the query plan while executing the query:

```plaintext
test
fedsql.execdirect
  method=true
  query="...FedSQL statements...";
quit;
```

Here is an example of how showStages is specified:

```plaintext
test
fedsql.execdirect
  showstages=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. Interfaces that are case-sensitive require you to specify the options exactly as documented.

---

FedSQL Query Walk-Through

Here is an example of a FedSQL query. Subsequent sections illustrate the outputs that are returned by the three query plan options.

---

The FedSQL Query

This FedSQL query specifies to join columns from two CAS tables and includes a calculated column in the combined result set. The source tables are named WorldCityCoords and WorldTemp. The calculated column is named AvgHighNation.
The request specifies a subquery to create the new column. For information to create the tables, see “WorldCityCoords” on page 154 and “WorldTemps” on page 155. The submission mode is excluded, so you can use the interface of your choice.

```sql
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;
```

### Method and validateOnly Output

Here is an example of the output when the ValidateOnly option (or NOEXEC) is specified with the Method option. When validateOnly (or NOEXEC) is specified with the Method option, the query is not executed.

#### 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 CASUSER(username).WORLDTEMPS
Sort
HashJoin (INNER)
SeqScan from CASUSER(username).WORLDTEMPS
SeqScan from CASUSER(username).WORLDTEMPS
```

The output is a high-level query plan.

### Method Output

Here is an example of the output when the Method option is used without validateOnly (or NOEXEC).
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 CASUSER(username).WORLDTEMPS
  Sort
    HashJoin (INNER)
    SeqScan from CASUSER(username).WORLDTEMPS
    SeqScan from CASUSER(username).WORLDTEMPS

Methods for stage 2
--------------------
  SubqueryScan
    Agg
      SeqScan with _pushed_ order by from CASUSER(username).WORLDTEMPS

Methods for stage 3
--------------------
  HashJoin (INNER)
    SeqScan from CASUSER(username).WORLDTEMPS
    SeqScan from CASUSER(username).WORLDTEMPS

Methods for stage 4
--------------------
  HashJoin (INNER)
    SeqScan from CASUSER(username).__fedsql_3__
    SeqScan from CASUSER(username).__fedsql_2__

Methods for stage 5
--------------------
  Sort
    SeqScan from CASUSER(username).__fedsql_4__

79 quit;

The output includes a brief summary of the execution stages in addition to the high-level query plan.

showStages Output

Here is an example of the output from the showStages option. The output can be long. For readability, each stage is presented as separate block.
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 CASUSER(username).WORLDTEMPS
Sort
HashJoin (INNER)
SeqScan from CASUSER(username).WORLDCITYCOORDS
SeqScan from CASUSER(username).WORLDTEMPS

Methods for stage 2
-------------------
SubqueryScan
Agg
SeqScan with _pushed order by from CASUSER(username).WORLDTEMPS

Stage query: create table "CASUSER(username)"."__fedsql_2__" [options replace=true replication=0 tableID=2] as select "AHN"."AVGHIGHNATION", "AHN"."Country" from (select "T1"."Country", AVG("T1"."AvgHigh") as "AVGHIGHNATION" from "CASUSER(username)"."WORLDTEMPS" [options tableID=1] T1 group by "T1"."Country")  "AHN"

Number of data access threads: 4 per worker

Input tables

CASUSER(username).WORLDTEMPS (rows 12, columns 4)
Partitioned with local sort of each partition
Time to open table: 0.000 seconds
Time to partition and sort: 0.016 seconds

Output table

CASUSER(username).__fedsql_2__ (rows 8, columns 2)

(stage 2) BEGIN 12:37:13 END 12:37:13 (0.115 seconds)

Like the other options, the query output begins with the high-level view of the plan nodes in the query plan. Then, it prints each stage query and query execution details for each stage.

The output of a showStages request begins with Stage 2. Stage 2 of this query plan processes the subquery in table WorldTemps first. In Stage 2, FedSQL performs an aggregate sort on column AV GHIGH using the values in column COUNTRY to create a new column named AVG_HIGH_NATION. Temporary table _fedsql_2__ is created to hold the results of the subquery.
Methods for stage 3
-------------------
HashJoin (INNER)
SeqScan from CASUSER(username).WORLDCITYCOORDS
SeqScan from CASUSER(username).WORLDTEMPS

Stage query:
create table "CASUSER(username)"."__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
"CASUSER(username)"."WORLDCITYCOORDS" {options tableID=1} T1 _hash_ inner join
"CASUSER(username)"."WORLDTEMPS" {options REPL=YES tableID=2} T2 on
("T1"."City"="T2"."City")

Number of data access threads: 4 per worker

Input tables

CASUSER(username).WORLDCITYCOORDS (rows 12, columns 4)
Time to open table: 0.000 seconds
Time to assign data to threads: 0.000 seconds

CASUSER(username).WORLDTEMPS (rows 12, columns 4)
Replicated to all workers
Time to open table: 0.000 seconds
Time to replicate: 0.000 seconds

Output table

CASUSER(username).__fedsql_3__ (rows 12, columns 6)

[stage 3] BEGIN 12:37:13 END 12:37:13 (0.084 seconds)

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.
Methods for stage 4
-------------------
HashJoin (INNER)
  SeqScan from CASUSER(username).__fedsql_3__
  SeqScan from CASUSER(username).__fedsql_2__

Stage query: create table "CASUSER(username)"."__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 "CASUSER(username)"."__fedsql_3__" {options tableID=2} T2 _hash_ inner join
"CASUSER(username)"."__fedsql_2__" {options REPL=YES tableID=1} T1 on
("T1"."Country"="T2"."Country")

Number of data access threads: 4 per worker

Input tables
CASUSER(username).__fedsql_2__ (rows 8, columns 2)
  Replicated to all workers
  Time to open table: 0.000 seconds
  Time to replicate: 0.000 seconds

CASUSER(username).__fedsql_3__ (rows 12, columns 6)
  Time to open table: 0.000 seconds
  Time to assign data to threads: 0.000 seconds

Output table
CASUSER(username).__fedsql_4__ (rows 12, columns 6)

[stage 4] BEGIN 12:37:13 END 12:37:13 (0.069 seconds)

In Stage 4, the plan joins temporary tables __fedsql_3__ and __fedsql_2__ to
create temporary table __fedsql_4__.

Methods for stage 5
-------------------
Sort
  SeqScan from CASUSER(username).__fedsql_4__

Stage query: select "T1"."City", "T1"."Country", "T1"."Latitude",
  "T1"."Longitude", "T1"."AVGHIGHCITY", "T1"."AVGHIGHNATION" from "CASUSER(username)"."__fedsql_4__"
{options REPL=YES tableID=1} T1 order by 2 collate linguistic (locale=en_US), 1 collate
linguistic (locale=en_US)

Number of data access threads: 1

Input tables
CASUSER(username).__fedsql_4__ (rows 12, columns 6)
  Replicated to one worker
  Time to open table: 0.000 seconds
  Time to replicate: 0.000 seconds

Output returned to client (rows 12, columns 6)

[stage 5] BEGIN 12:37:13 END 12:37:13 (0.300 seconds)

[fedSql.execDirect] BEGIN 12:37:13 END 12:37:13 (0.568 seconds)
Finally, stage 5 performs a sort and sequential scan to display the contents of temporary table _fedsql_4_

---

**Optimizing FedSQL Performance by Modifying the FedSQL Query Plan**

**Cntl Option**

The execDirect action’s Cntl option enables you to control aspects of query execution. This Cntl option supports 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.

- **dynamicCardinality=true | false**
  The FedSQL query planner does not perform cardinality estimations before selecting a query plan (dynamicCardinality=false). Cardinality estimation can improve the accuracy of selectivity estimates for join conditions and WHERE clause predicates. Improved accuracy can lead to better join order decisions and faster query execution times for some queries. However, dynamic cardinality does not help all queries and can slow performance in the query planning process. For information about the types of queries that can benefit from a cardinality estimation, see “Using the Dynamic Cardinality Instruction” on page 25.

  **Note:** This option is available beginning with SAS Viya 3.5.

- **optimizeVarbinaryPrecision=true | false**
  The fedSql.execDirect action uses the declared precision for columns of the VARBINARY data type by default (optimizeVarbinaryPrecision=false). Setting optimizeVarbinaryPrecision=true optimizes VARBINARY precision by using a precision that is appropriate to the actual data, instead of the declared precision. The greatest benefits from this option are achieved when the declared precision is far larger than the precision of the actual data. Then, this option can improve performance, reduce the memory footprint, and create VARBINARY columns in new tables with the needed size rather than propagating the precision from the source table.

  **Note:** The optimizeVarbinaryPrecision= instruction can be turned on at the server level by setting an environment variable. The Cntl option setting overrides the value set in the environment variable. For information about the FedSQL environment variables for CAS, see SAS Viya Administration: SAS Cloud Analytic Services.
optimizeVarcharPrecision=true | false
The fedSql.execDirect action uses the declared precision for columns of the VARCHAR data type by default (optimizeVarcharPrecision=false). Setting optimizeVarcharPrecision=true optimizes VARCHAR precision by using a precision that is appropriate to the actual data, instead of the declared precision. The greatest benefits from this option are achieved when the declared precision is far larger than the precision of the actual data. Then, this option can improve performance, reduce the memory footprint, and create VARCHAR columns in new tables with the needed size rather than propagating the precision from the source table.

Note: This option is available beginning with SAS Viya 3.5.

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.

Setting preserveJoinOrder=true instructs the query optimizer to join the tables in the order specified.

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

requireFullPassThrough=true | false
The execDirect action automatically 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.
How to Specify the Cntl Instructions

In PROC FEDSQL, Cntl is a procedure option. Instructions are specified within parenthesis. The value true is implied by the mention of an instruction. Omission of an option means false (the default value).

Here is an example of how the instructions are specified in the FEDSQL procedure:

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

Multiple instructions are separated by a space.

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

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

```sas
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.

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

Using Cntl Instructions That Affect FedSQL Implicit Pass-Through

FedSQL implicit pass-through is enabled by default. Consider disabling pass-through in these situations:

- FedSQL processing is faster in CAS than in the data source.
- A Cartesian product is being produced, and the cost of pulling all of the participating tables into CAS is less than pulling the final result set into CAS.

You might want to use requireFullPassThrough for FedSQL for the opposite situation: requests that require a lot of subsetting, which is typically done more efficiently by the data source.
Using the Dynamic Cardinality Instruction

Cardinality refers to the uniqueness of the data values that are contained in a column. High cardinality means that a column contains a large percentage of unique values; low cardinality means that a column contains a small percentage of unique values.

The FedSQL Query Planner for CAS does not examine the cardinality of input columns by default as knowledge of cardinality is not helpful for all queries. When the dynamicCardinality instruction is enabled in the Cntl option, the execDirect action performs cardinality estimations by calling the CAS highCardinality action from the dataPreprocess action set. The dataPreprocess.highCardinality action reads a sample of the input data to estimate the number of distinct values in each column. Calling this action can slow performance in the query planning process and not all queries benefit from the results of a cardinality estimation.

Consider enabling dynamic cardinality for queries that have the following characteristics:

- The query joins three or more tables.
- One or more tables in the query contain tens to hundreds of millions of rows, and the other tables are smaller dimension tables.

In general, the potential value of dynamic cardinality increases as the number of tables being joined increases and as data size increases. Dynamic cardinality does not help queries that join fewer than three tables.

When working with very large tables, a sound join order decision can avoid intermediate result sets that do not fit into memory. A sound join order decision can also result in opportunities to broadcast small intermediate result tables instead of auto-partitioning one or more very large tables. Memory and time requirements for partitioning large tables can be high.

Currently, cardinality estimations gathered during one query are not stored for use by later queries.
Joining Data with FedSQL

Overview of Joins ................................................. 28

Example: Typical Two-Table Join .................................. 30
  Details .......................................................... 30
  Program ......................................................... 30

Example: Typical Three-Table Join ................................ 31
  Details .......................................................... 31
  Program ......................................................... 31

Example: Simple Join Including All Columns ....................... 32
  Details .......................................................... 32
  Program ......................................................... 32

Example: Equijoin Including All Columns ......................... 33
  Details .......................................................... 33
  Program ......................................................... 33

Example: Simple Cross Join ....................................... 34
  Details .......................................................... 34
  Program ......................................................... 34

Example: Cross Join with Specified Columns and a WHERE Clause 36
  Details .......................................................... 36
  Program ......................................................... 36

Example: Qualified Join with an ON Clause ....................... 37
  Details .......................................................... 37
  Program ......................................................... 37
  Related Information ........................................... 38

Example: Qualified Join with a USING Clause .................... 38
  Details .......................................................... 38
  Program ......................................................... 38
  Related Information ........................................... 39

Example: Qualified Join with an ON Clause and a WHERE Clause 40
  Details .......................................................... 40
  Program ......................................................... 40
  Related Information ........................................... 40

Example: Natural Join ............................................ 41
  Details .......................................................... 41
  Program ......................................................... 41
  Related Information ........................................... 42

Example: Natural Join with a WHERE Clause .................... 42
  Details .......................................................... 42
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 specified with either a WHERE clause or an ON 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. The join criteria for an inner join is
specified with an ON 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. The join criteria for an inner join is
specified with an ON 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. In
addition, it includes 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. In
addition, it includes 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.
Example: Typical Two-Table Join

Details

This example joins a column from two tables to produce a single result set.

Program

```sql
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**

<table>
<thead>
<tr>
<th>PRODUCT</th>
<th>TOTALS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corn</td>
<td>$555,789</td>
</tr>
<tr>
<td>Wheat</td>
<td>$781,183</td>
</tr>
<tr>
<td>Wheat</td>
<td>$2,789,654</td>
</tr>
<tr>
<td>Rice</td>
<td>$189,400</td>
</tr>
<tr>
<td>Barley</td>
<td>$899,453</td>
</tr>
</tbody>
</table>

Key Ideas

- Most join operations contain at least one join condition, which is specified with either a WHERE clause or an ON clause. This example specifies a WHERE clause.
- The query selects a column from each input table (Product from table “Products” on page 152 and Totals from table “Sales” on page 153) 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 query syntax includes a WHERE clause that specifies an equality condition, this is in equijoin.
Example: Typical Three-Table Join

Details

This example joins a column from three tables to produce a single result set.

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

<table>
<thead>
<tr>
<th>PRODUCT</th>
<th>TOTALS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corn</td>
<td>$555,789</td>
<td>Nagasaki</td>
</tr>
<tr>
<td>Wheat</td>
<td>$781,183</td>
<td>Tokyo</td>
</tr>
<tr>
<td>Wheat</td>
<td>$2,789,654</td>
<td>Little Rock</td>
</tr>
<tr>
<td>Rice</td>
<td>$189,400</td>
<td>Boulder</td>
</tr>
<tr>
<td>Barley</td>
<td>$899,453</td>
<td>Buenos Aires</td>
</tr>
</tbody>
</table>

Key Ideas

- This FedSQL SELECT statement specifies a column from each of three tables: Product from table “Products” on page 152, Totals from table “Sales” on page 153, and City from table “Customers” on page 145 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 merges the content of the Products and Sales tables based on the values in a common column Prodid. The content of the result set is then merged with table Customers based on the value in a CustId column that the Sales and Customers tables have in common.

- Because the query syntax includes a WHERE clause that specifies an equality condition, this is in equijoin.
Example: Simple Join Including All Columns

Details

This example joins all columns from the Products table with all columns from the Sales table into a single result set.

Program

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

Here is the output from the SELECT statement:

**Output 3.3** Simple Join of Tables Products and Sales

<table>
<thead>
<tr>
<th>PRODID</th>
<th>PRODUCT</th>
<th>PRODID</th>
<th>CUSTID</th>
<th>TOTALS</th>
<th>COUNTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>3421</td>
<td>Wheat</td>
<td>3421</td>
<td>4</td>
<td>$781,183</td>
<td>Japan</td>
</tr>
<tr>
<td>3421</td>
<td>Wheat</td>
<td>3234</td>
<td>1</td>
<td>$169,400</td>
<td>United States</td>
</tr>
<tr>
<td>3421</td>
<td>Wheat</td>
<td>3975</td>
<td>5</td>
<td>$899,453</td>
<td>Argentina</td>
</tr>
<tr>
<td>3421</td>
<td>Wheat</td>
<td>1424</td>
<td>3</td>
<td>$555,780</td>
<td>Japan</td>
</tr>
<tr>
<td>3421</td>
<td>Wheat</td>
<td>3421</td>
<td>2</td>
<td>$2,789,654</td>
<td>United States</td>
</tr>
<tr>
<td>3975</td>
<td>Barley</td>
<td>3421</td>
<td>5</td>
<td>$899,453</td>
<td>Argentina</td>
</tr>
<tr>
<td>3975</td>
<td>Barley</td>
<td>3975</td>
<td>4</td>
<td>$781,183</td>
<td>Japan</td>
</tr>
<tr>
<td>3975</td>
<td>Barley</td>
<td>3234</td>
<td>1</td>
<td>$169,400</td>
<td>United States</td>
</tr>
<tr>
<td>3975</td>
<td>Barley</td>
<td>1424</td>
<td>3</td>
<td>$555,780</td>
<td>Japan</td>
</tr>
<tr>
<td>3975</td>
<td>Barley</td>
<td>3421</td>
<td>2</td>
<td>$2,789,654</td>
<td>United States</td>
</tr>
<tr>
<td>1424</td>
<td>Corn</td>
<td>1424</td>
<td>3</td>
<td>$555,780</td>
<td>Japan</td>
</tr>
<tr>
<td>1424</td>
<td>Corn</td>
<td>3234</td>
<td>1</td>
<td>$169,400</td>
<td>United States</td>
</tr>
<tr>
<td>1424</td>
<td>Corn</td>
<td>3421</td>
<td>4</td>
<td>$781,183</td>
<td>Japan</td>
</tr>
<tr>
<td>1424</td>
<td>Corn</td>
<td>3975</td>
<td>5</td>
<td>$899,453</td>
<td>Argentina</td>
</tr>
<tr>
<td>1424</td>
<td>Corn</td>
<td>3421</td>
<td>2</td>
<td>$2,789,654</td>
<td>United States</td>
</tr>
<tr>
<td>3422</td>
<td>Oat</td>
<td>3421</td>
<td>4</td>
<td>$781,183</td>
<td>Japan</td>
</tr>
<tr>
<td>3422</td>
<td>Oat</td>
<td>3234</td>
<td>1</td>
<td>$169,400</td>
<td>United States</td>
</tr>
<tr>
<td>3422</td>
<td>Oat</td>
<td>3975</td>
<td>5</td>
<td>$899,453</td>
<td>Argentina</td>
</tr>
<tr>
<td>3422</td>
<td>Oat</td>
<td>1424</td>
<td>3</td>
<td>$555,780</td>
<td>Japan</td>
</tr>
<tr>
<td>3234</td>
<td>Rice</td>
<td>3234</td>
<td>1</td>
<td>$169,400</td>
<td>United States</td>
</tr>
<tr>
<td>3234</td>
<td>Rice</td>
<td>3421</td>
<td>4</td>
<td>$781,183</td>
<td>Japan</td>
</tr>
<tr>
<td>3234</td>
<td>Rice</td>
<td>3975</td>
<td>5</td>
<td>$899,453</td>
<td>Argentina</td>
</tr>
</tbody>
</table>
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 152 and "Sales" on page 153, 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.

Example: Equijoin Including All Columns

Details

This example joins all columns from tables Products and Sales into a single result set based on an equality condition.

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**

<table>
<thead>
<tr>
<th>PRODID</th>
<th>PRODUCT</th>
<th>PRODID</th>
<th>CUSTID</th>
<th>TOTALS</th>
<th>COUNTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1424</td>
<td>Corn</td>
<td>1424</td>
<td>3</td>
<td>$555,789</td>
<td>Japan</td>
</tr>
<tr>
<td>3234</td>
<td>Rice</td>
<td>3234</td>
<td>1</td>
<td>$189,400</td>
<td>United States</td>
</tr>
<tr>
<td>3421</td>
<td>Wheat</td>
<td>3421</td>
<td>4</td>
<td>$781,183</td>
<td>Japan</td>
</tr>
<tr>
<td>3421</td>
<td>Wheat</td>
<td>3421</td>
<td>2</td>
<td>$2,789,654</td>
<td>United States</td>
</tr>
<tr>
<td>3975</td>
<td>Barley</td>
<td>3975</td>
<td>5</td>
<td>$899,453</td>
<td>Argentina</td>
</tr>
</tbody>
</table>

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 152 and “Sales” on page 153 where the values match for the column Prodid, which exists in both tables. Because the 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 would not be duplicated.

Example: Simple Cross Join

Details

This example uses cross join syntax to merge all columns from tables Products and Sales into a single result set.

Program

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

Here is the output from the SELECT statement:
**Cross Join of Two Tables**

**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. It produces the same results as a simple join of the two tables.
Example: Cross Join with Specified Columns and a WHERE Clause

Details

This example uses cross join syntax and a WHERE clause to merge specified columns from tables Products and Sales into a single result set. The result set is filtered with 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**

<table>
<thead>
<tr>
<th>PRODID</th>
<th>PRODUCT</th>
<th>TOTALS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1424</td>
<td>Corn</td>
<td>$555,769</td>
</tr>
<tr>
<td>3421</td>
<td>Wheat</td>
<td>$781,183</td>
</tr>
<tr>
<td>3421</td>
<td>Wheat</td>
<td>$2,789,654</td>
</tr>
<tr>
<td>3234</td>
<td>Rice</td>
<td>$189,400</td>
</tr>
<tr>
<td>3975</td>
<td>Barley</td>
<td>$899,453</td>
</tr>
</tbody>
</table>

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 152 and column Totals from "Sales" on page 153. 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.
Example: Qualified Join with an ON Clause

Details

This example uses join syntax with the ON clause to merge all columns from tables Products and Sales into a single result set.

Program

```sql
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

<table>
<thead>
<tr>
<th>PRODID</th>
<th>PRODUCT</th>
<th>PRODID</th>
<th>CUSTID</th>
<th>TOTALS</th>
<th>COUNTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1424</td>
<td>Corn</td>
<td>3234</td>
<td>1</td>
<td>$189,400</td>
<td>United States</td>
</tr>
<tr>
<td>1424</td>
<td>Corn</td>
<td>3421</td>
<td>2</td>
<td>$2,789,654</td>
<td>United States</td>
</tr>
<tr>
<td>3421</td>
<td>Wheat</td>
<td>3234</td>
<td>1</td>
<td>$189,400</td>
<td>United States</td>
</tr>
<tr>
<td>3421</td>
<td>Wheat</td>
<td>3421</td>
<td>2</td>
<td>$2,789,654</td>
<td>United States</td>
</tr>
<tr>
<td>3422</td>
<td>Oat</td>
<td>3421</td>
<td>2</td>
<td>$2,789,654</td>
<td>United States</td>
</tr>
<tr>
<td>3422</td>
<td>Oat</td>
<td>3234</td>
<td>1</td>
<td>$189,400</td>
<td>United States</td>
</tr>
<tr>
<td>3234</td>
<td>Rice</td>
<td>3234</td>
<td>1</td>
<td>$189,400</td>
<td>United States</td>
</tr>
<tr>
<td>3234</td>
<td>Rice</td>
<td>3421</td>
<td>2</td>
<td>$2,789,654</td>
<td>United States</td>
</tr>
<tr>
<td>3975</td>
<td>Barley</td>
<td>3234</td>
<td>1</td>
<td>$189,400</td>
<td>United States</td>
</tr>
<tr>
<td>3975</td>
<td>Barley</td>
<td>3421</td>
<td>2</td>
<td>$2,789,654</td>
<td>United States</td>
</tr>
</tbody>
</table>

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 152 and “Sales” on page 153. 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

- “Understanding Inner and Outer Join Types” on page 43
- “Example: Left Outer Qualified Join” on page 44
- “Example: Right Outer Qualified Join” on page 46
- “Example: Full Outer Qualified Join” on page 48

Example: Qualified Join with a USING Clause

Details

This example uses join syntax with the USING clause to merge all columns from tables Products and Sales into a single result set.

Program

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

Here is the output from the SELECT statement:
### 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 152 and “Sales” on page 153. 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

- “Understanding Inner and Outer Join Types” on page 43
Example: Qualified Join with an ON Clause and a WHERE Clause

Details
This example uses join syntax and an ON clause to join specified columns from tables Products and Sales into a single result set. The result set is filtered with 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**

<table>
<thead>
<tr>
<th>PRODID</th>
<th>PRODUCT</th>
<th>TOTALS</th>
</tr>
</thead>
<tbody>
<tr>
<td>3234</td>
<td>Rice</td>
<td>$189,400</td>
</tr>
<tr>
<td>3234</td>
<td>Rice</td>
<td>$2,789,654</td>
</tr>
</tbody>
</table>

Key Idea

This qualified join example selects columns Prodid and Product from table “Products” on page 152 and column Totals from table “Sales” on page 153. 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
- “Understanding Inner and Outer Join Types” on page 43
Example: Natural Join

Details

This example uses natural join syntax to merge the content in tables Products and Sales into a single result set.

Program

```sql
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**

<table>
<thead>
<tr>
<th>PRODID</th>
<th>PRODUCT</th>
<th>CUSTID</th>
<th>TOTALS</th>
<th>COUNTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1424</td>
<td>Corn</td>
<td>3</td>
<td>$555,789</td>
<td>Japan</td>
</tr>
<tr>
<td>3234</td>
<td>Rice</td>
<td>1</td>
<td>$189,400</td>
<td>United States</td>
</tr>
<tr>
<td>3421</td>
<td>Wheat</td>
<td>4</td>
<td>$781,183</td>
<td>Japan</td>
</tr>
<tr>
<td>3421</td>
<td>Wheat</td>
<td>2</td>
<td>$2,789,654</td>
<td>United States</td>
</tr>
<tr>
<td>3975</td>
<td>Barley</td>
<td>5</td>
<td>$899,453</td>
<td>Argentina</td>
</tr>
</tbody>
</table>

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 152 and “Sales” on page 153. 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

- “Understanding Inner and Outer Join Types” on page 43

Example: Natural Join with a WHERE Clause

Details

This example uses natural join syntax to merge specified columns from the Customers and Sales tables. The result set is filtered with a WHERE clause.

Program

```sql
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**

<table>
<thead>
<tr>
<th>CITY</th>
<th>TOTALS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Little Rock</td>
<td>$2,789,654</td>
</tr>
<tr>
<td>Boulder</td>
<td>$189,400</td>
</tr>
</tbody>
</table>

Key Idea

This natural join example selects columns City and Totals from the tables “Sales” on page 153 and “Customers” on page 145. 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.
Understanding 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 join 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: Inner Join

Details

This example uses inner join syntax to merge columns from tables Customers and Sales into a single result set.

Program

```sql
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: Left Outer Qualified Join

Details

This example uses left outer join syntax to merge specified columns from tables Customers and Sales into a single result set.

Program

```sql
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

<table>
<thead>
<tr>
<th>CITY</th>
<th>TOTALS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tokyo</td>
<td></td>
</tr>
<tr>
<td>Boulder</td>
<td>$189,400</td>
</tr>
<tr>
<td>Boulder</td>
<td>$555,789</td>
</tr>
<tr>
<td>Boulder</td>
<td>$781,183</td>
</tr>
<tr>
<td>Boulder</td>
<td>$2,789,654</td>
</tr>
<tr>
<td>Boulder</td>
<td>$899,453</td>
</tr>
<tr>
<td>Little Rock</td>
<td>$555,789</td>
</tr>
<tr>
<td>Little Rock</td>
<td>$189,400</td>
</tr>
<tr>
<td>Little Rock</td>
<td>$781,183</td>
</tr>
<tr>
<td>Little Rock</td>
<td>$2,789,654</td>
</tr>
<tr>
<td>Little Rock</td>
<td>$899,453</td>
</tr>
<tr>
<td>Nagasaki</td>
<td></td>
</tr>
<tr>
<td>Buenos Aires</td>
<td></td>
</tr>
</tbody>
</table>
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: Left Outer Natural Join

Details

This example uses natural left outer join syntax to merge tables Products and Sales into a single result set.

Program

```sql
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**

<table>
<thead>
<tr>
<th>PRODID</th>
<th>CUSTID</th>
<th>TOTALS</th>
<th>COUNTRY</th>
<th>PRODUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1424</td>
<td>3</td>
<td>$555,789</td>
<td>Japan</td>
<td>Corn</td>
</tr>
<tr>
<td>3421</td>
<td>4</td>
<td>$781,183</td>
<td>Japan</td>
<td>Wheat</td>
</tr>
<tr>
<td>3421</td>
<td>2</td>
<td>$2,789,654</td>
<td>United States</td>
<td>Wheat</td>
</tr>
<tr>
<td>3234</td>
<td>1</td>
<td>$189,400</td>
<td>United States</td>
<td>Rice</td>
</tr>
<tr>
<td>3975</td>
<td>5</td>
<td>$899,453</td>
<td>Argentina</td>
<td>Barley</td>
</tr>
</tbody>
</table>

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 joins the tables on matching values in 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: Right Outer Qualified Join

Details

This example uses right outer join syntax to merge tables Products and Sales into a single result set.

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**

<table>
<thead>
<tr>
<th>PRODID</th>
<th>PRODUCT</th>
<th>PRODID</th>
<th>CUSTID</th>
<th>TOTALS</th>
<th>COUNTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>3422</td>
<td>Oat</td>
<td>3421</td>
<td>2</td>
<td>2,789,654</td>
<td>United States</td>
</tr>
<tr>
<td>3234</td>
<td>Rice</td>
<td>3421</td>
<td>2</td>
<td>2,789,654</td>
<td>United States</td>
</tr>
<tr>
<td>1424</td>
<td>Corn</td>
<td>3421</td>
<td>2</td>
<td>2,789,654</td>
<td>United States</td>
</tr>
<tr>
<td>3421</td>
<td>Wheat</td>
<td>3421</td>
<td>2</td>
<td>2,789,654</td>
<td>United States</td>
</tr>
<tr>
<td>3975</td>
<td>Barley</td>
<td>3421</td>
<td>2</td>
<td>2,789,654</td>
<td>United States</td>
</tr>
<tr>
<td>3234</td>
<td>Rice</td>
<td>3234</td>
<td>1</td>
<td>189,400</td>
<td>United States</td>
</tr>
<tr>
<td>1424</td>
<td>Corn</td>
<td>3234</td>
<td>1</td>
<td>189,400</td>
<td>United States</td>
</tr>
<tr>
<td>3421</td>
<td>Wheat</td>
<td>3234</td>
<td>1</td>
<td>189,400</td>
<td>United States</td>
</tr>
<tr>
<td>3422</td>
<td>Oat</td>
<td>3234</td>
<td>1</td>
<td>189,400</td>
<td>United States</td>
</tr>
<tr>
<td>3975</td>
<td>Barley</td>
<td>3234</td>
<td>1</td>
<td>189,400</td>
<td>United States</td>
</tr>
<tr>
<td>1424</td>
<td></td>
<td>1424</td>
<td>3</td>
<td>555,789</td>
<td>Japan</td>
</tr>
<tr>
<td>3421</td>
<td></td>
<td>3421</td>
<td>4</td>
<td>781,183</td>
<td>Japan</td>
</tr>
<tr>
<td>3975</td>
<td></td>
<td>3975</td>
<td>5</td>
<td>899,453</td>
<td>Argentina</td>
</tr>
</tbody>
</table>

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: Right Outer Natural Join**

**Details**

This example uses natural right outer join syntax to merge tables `Products` and `Sales` into a single result set.

**Program**

```sql
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*

<table>
<thead>
<tr>
<th>PRODID</th>
<th>PRODUCT</th>
<th>CUSTID</th>
<th>TOTALS</th>
<th>COUNTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>3234</td>
<td>Rice</td>
<td>1</td>
<td>$189,400</td>
<td>United States</td>
</tr>
<tr>
<td>3975</td>
<td>Barley</td>
<td>5</td>
<td>$699,453</td>
<td>Argentina</td>
</tr>
<tr>
<td>1424</td>
<td>Corn</td>
<td>3</td>
<td>$555,789</td>
<td>Japan</td>
</tr>
<tr>
<td>3421</td>
<td>Wheat</td>
<td>4</td>
<td>$781,183</td>
<td>Japan</td>
</tr>
<tr>
<td>3421</td>
<td>Wheat</td>
<td>2</td>
<td>$2,789,654</td>
<td>United States</td>
</tr>
</tbody>
</table>

**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 row values that match for the column `Prodid`. The result set also includes a row from the `Sales` table that does
Example: Full Outer Qualified Join

Details

This example uses full outer join syntax to merge tables Products and Sales into a single result set.

Program

```sql
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

<table>
<thead>
<tr>
<th>PRODID</th>
<th>PRODUCT</th>
<th>PRODID</th>
<th>CUSTID</th>
<th>TOTALS</th>
<th>COUNTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>3234</td>
<td>Rice</td>
<td>3234</td>
<td>1</td>
<td>$189,400</td>
<td>United States</td>
</tr>
<tr>
<td>3234</td>
<td>Rice</td>
<td>1424</td>
<td>3</td>
<td>$555,789</td>
<td>Japan</td>
</tr>
<tr>
<td>3234</td>
<td>Rice</td>
<td>3421</td>
<td>4</td>
<td>$781,183</td>
<td>Japan</td>
</tr>
<tr>
<td>3234</td>
<td>Rice</td>
<td>3421</td>
<td>2</td>
<td>$2,789,654</td>
<td>United States</td>
</tr>
<tr>
<td>3234</td>
<td>Rice</td>
<td>3975</td>
<td>5</td>
<td>$899,453</td>
<td>Argentina</td>
</tr>
<tr>
<td>1424</td>
<td>Corn</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>3421</td>
<td>Wheat</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>3422</td>
<td>Oat</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>3975</td>
<td>Barley</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

Key Ideas

- A full outer join is requested with the syntax FULL [OUTER]. A full outer join preserves non-matching rows from both tables. That is, a full outer join returns all matching and non-matching rows from the left and right table.
- This qualified join example returns matching rows that have the value Rice in the Product column. The result set also includes all rows from both
Example: Full Outer Natural Join

Details

This example uses natural full outer join syntax to merge tables Products and Sales into a single result set.

Program

```sql
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**

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

Key Ideas

- A full outer join preserves unmatched rows from both tables. That is, a full outer join returns all matching and non-matching rows from the left and right table.
- This natural join example returns rows that have matching values for the column Prodid. The result set also includes a row from the Sales table and a row from the Products table that do not match the join condition. As a full outer join, all rows from both tables are returned.
Query Expressions and Subqueries

Overview of Query Expressions

A query expression or query consists of one or more SELECT statements that produce a result set. One way that multiple SELECT statements can be combined is by using subqueries. Another is by using set operators.

Beginning in SAS Viya 3.5, FedSQL supports the UNION set operator on the CAS server. The UNION set operator combines the result of two queries into a single result set that contains the unique rows from both queries.

By default, columns are combined by position in the tables, regardless of the column names. Columns in the same relative position must have compatible data types and lengths and each query must have the same number of columns for the set operation to succeed. The UNION operator supports optional keywords to change the default behavior.
Keyword | Purpose
---|---
ALL | returns all rows from both queries, including duplicates, instead of only unique rows.
CORRESPONDING | combines columns from the queries by column name instead of by position. The CORRESPONDING keyword returns rows for all columns that have matching names. Any columns whose names do not match are ignored.
BY | submits a column list to the CORRESPONDING keyword and returns rows only for the columns in the list.

A query expression with the UNION operator is evaluated as follows:

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

### Examples of Query Expressions with the UNION Set Operator

To understand how a query expression that specifies the UNION operator works, consider the following examples. The examples operate on three tables, named Numbers1, Numbers2, and Numbers3. The following outputs show the content of the tables:

<table>
<thead>
<tr>
<th>Numbers1 Table</th>
<th>Numbers2 Table</th>
<th>Numbers3 Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
<td>Z</td>
</tr>
<tr>
<td>one</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>three</td>
<td>3</td>
<td>30</td>
</tr>
<tr>
<td>four</td>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>four</td>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>five</td>
<td>5</td>
<td>50</td>
</tr>
</tbody>
</table>
Tables Numbers1 and Numbers2 contain columns of the same names and data types, ordered in the same positions. Table Numbers3 contains columns of the same names and content, but they are in a different position than the columns in tables Numbers1 and Numbers2. For information about creating the tables, see Appendix 2, “Tables Used in Examples,” on page 143.

Combining Queries with the UNION Operator: Example of the Default Behavior

Details

The following example combines tables Numbers1 and Numbers2 with the UNION operator. The query does not include any optional keywords.

Program

```sql
select * from Numbers1 union select * from Numbers2;
```

The result set contains the unique rows from both input tables.

Output 4.1  Output from the UNION of Tables Numbers1 and Numbers2

```
<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>five</td>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>four</td>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>one</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>three</td>
<td>3</td>
<td>30</td>
</tr>
<tr>
<td>two</td>
<td>2</td>
<td>20</td>
</tr>
</tbody>
</table>
```

Key Ideas

- The request succeeds because the columns from the intermediate result sets are in the same position and have the same general data type. The UNION operator matches a CHAR column with a VARCHAR column, and it matches an integer type with another integer type. However, it does not match a CHAR type with an integer type or with a DOUBLE.

- The column names and column lengths in the intermediate (and final) result sets are the same. When the column names and column lengths are different, the UNION operator uses the column names and column sizes of the larger column.

- To include all rows (including duplicates), you can specify the ALL keyword in the request.
Example: Combining Tables That Have Columns in Different Positions

Details
The following example combines tables Numbers1 and Numbers3 with the UNION operator. The request specifies the CORRESPONDING keyword.

Program
```
select * from Numbers1 union corresponding select * from Numbers3;
```
The result set contains the unique rows from all columns that have matching names.

Output 4.2 Output from the UNION CORRESPONDING Query

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>five</td>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>four</td>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>one</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>seven</td>
<td>7</td>
<td>70</td>
</tr>
<tr>
<td>six</td>
<td>6</td>
<td>60</td>
</tr>
<tr>
<td>three</td>
<td>3</td>
<td>30</td>
</tr>
</tbody>
</table>

Key Ideas
- The CORRESPONDING keyword uses the column names of the input tables to match columns, instead of column position. Without the CORRESPONDING keyword, this UNION request would have failed.
- As with the default UNION behavior, the result set retains the column order and column sizes from the larger column. In this example, the result set uses the column order and column sizes of table Numbers1.
- The CORRESPONDING keyword returns all columns from both queries that have matching names. Columns whose names do not match are ignored. This enables you to combine intermediate result sets of different sizes. To combine rows from only one or a few columns in the intermediate result sets, you can use the BY keyword with the CORRESPONDING keyword.
- The result set contains the unique rows from both input tables. To include all rows from the intermediate result sets in the output, you can specify the ALL keyword in the request.
Example: Specifying the Columns to Include in the UNION Result Set

Details
The following example combines tables Numbers1 and Numbers3 and specifies the CORRESPONDING and BY keywords.

Program

```
select * from Numbers1 union corresponding by (x)
``` select * from Numbers3;

The result set contains the unique rows from the column indicated by the BY keyword only.

Output 4.3  Output from UNION CORRESPONDING BY Query

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>five</td>
</tr>
<tr>
<td>four</td>
</tr>
<tr>
<td>one</td>
</tr>
<tr>
<td>seven</td>
</tr>
<tr>
<td>six</td>
</tr>
<tr>
<td>three</td>
</tr>
</tbody>
</table>

Key Ideas

- The CORRESPONDING keyword specifies to match columns by column name, instead of by position.
- The BY keyword supplies the name or names of the column or columns to be returned.
- The columns in the BY column list must exist in both intermediate result sets.
- The BY keyword does not override the asterisk in the SELECT queries. The intermediate result sets that are created by the SELECT statements continue to be created with all columns from the tables. When the BY keyword is encountered, columns that do not match are discarded.
- The result set contains unique instances of the rows from both input tables. To include all rows (including duplicates) in the result set, you can specify the ALL keyword.
Subqueries

Overview of Subqueries

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

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

Key Ideas

- *Something* is a "<sql-expression> 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> 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 is to put a join inside the FROM clause subquery that specifies calculated values in the SELECT list. The outer query can then group the results by the calculated values. Here is an example of such a query. The subquery specifies the SUBSTRING function to create the intermediate result set.

```sql
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. The columns are stored in 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 WORLD_CITYCOORDS 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.

```sql
select
  C.*, T.AvgHigh as AvgHighCity, AvgHighNation
from world_citycoords 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 4.4 Results of Query on Tables WorldCityCoords and WorldTemps

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

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 and the content of the output variable AHN based on the values of the columns City and Country. The WorldCityCoords and WorldTemps tables have the City column in common. Table WorldTemps and output variable AHN have the Country column in common.
- The outer query orders the results of the equijoin by City and Country.

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-value-constructor, row-value-constructor, …)
- ROW (row-value-constructor, row-value-constructor, …)
- 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. The ROW keyword is optional. This example illustrates the use of the ROW keyword with a row value constructor:

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

The example could be written as follows:

```sql
select * from WorldTemps where (city, country) = ('Madrid', 'Spain')
```
Chapter 5
FedSQL Language Fundamentals ........................................ 63

Chapter 6
FedSQL Expressions and Predicates .................................... 75

Chapter 7
FedSQL Formats .............................................................. 99

Chapter 8
FedSQL Functions .......................................................... 107

Chapter 9
FedSQL Statements ......................................................... 113

Chapter 10
FedSQL Table Options ..................................................... 131
Locale

The locale identifies the language and possibly a regional dialect to use for the user interface. TheFedSQL.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 collating sequence indicated in the COLLATE= CAS session option. The default value, COLLATE="UCA", requests a locale-appropriate collating 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.
### Table 5.1 FedSQL Data Type Translation for CAS Tables

<table>
<thead>
<tr>
<th>FedSQL Data Type</th>
<th>CAS Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT¹</td>
<td>INT64</td>
<td>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.</td>
</tr>
<tr>
<td>CHAR((n))</td>
<td>CHAR((n))</td>
<td>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.</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DOUBLE</td>
<td>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.</td>
</tr>
</tbody>
</table>
### FedSQL Data Type | CAS Data Type | Description
--- | --- | ---
INTEGER\(^1\) | 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.

VARBINARY\(^2\) | VARBINARY | Varying-length binary opaque data. This data type is used to store image data, audio, documents, and other unstructured data.

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 \(\text{varchar}(10)\) is specified and the character string is only five characters long, only five characters are stored in the column.

---

1 Support for the integer types begins with SAS Viya 3.3.
2 Support for VARBINARY begins with SAS Viya 3.5. In the initial release, you can read and write VARBINARY columns in CAS tables. However, VARBINARY columns are not displayed by the FedSQL SELECT statement. The SELECT statement will display a value for a VARBINARY column if you use the PUT function to apply a CAS format on the column. For example, you might apply the $HEX. format on a VARBINARY columns to display the content in hexadecimal notation.

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. A SAS time format is applied to time values. A SAS date format is applied to date values. A SAS datetime format is applied to timestamp values.

CAS tables use the UTF-8 character set.
Identifiers

Overview of Identifiers

When displaying column information and creating new tables from existing tables in CAS, FedSQL preserves the identifiers in the input tables. FedSQL preserves the casing of the identifiers, but it is not case-sensitive. That is, the output of the SELECT statement might display the stored column names as uppercase, lowercase, or mixed case, depending on how the column names are stored. However, when column names are compared for a join, the comparison is case-insensitive. The resulting join output column matches the casing of the column from the first table that is specified in the join.

For referencing columns in FedSQL statements and for defining column aliases, 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 69.

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 7, “FedSQL Formats,” on page 99.

Handling of Nonexistent Data

FedSQL for CAS treats null and nonexistent values in CHAR, DOUBLE, and VARCHAR columns as SAS missing values. A SAS missing value is a value that is known to SAS and can be counted.

FedSQL treats null and nonexistent values in INT64, INT32, VARBINARY, and DOUBLE columns that were previously converted from ANSI DATE, TIME, and TIMESTAMP data types as ANSI null values. The columns are stored as having a zero length, as described in the CAS documentation. However, in ANSI SQL, nulls and nonexistent data have no data value. That is, nulls are treated as unknown values.

The use of missing values has implications for query processing, particularly in a WHERE clause, HAVING clause, or an outer join ON clause. Because it is a known value, a SAS missing value can pass both an equality test (WHERE column-value = NULL) and an existence test (WHERE column-value is NULL). Because it is not known, an ANSI null value fails an equality test. However, it succeeds in an existence test.

The following table summarizes the differences and similarities between SAS missing values and ANSI null values.

<table>
<thead>
<tr>
<th>Attribute or Behavior</th>
<th>ANSI Null Values</th>
<th>SAS Missing Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>internal representation</td>
<td>metadata</td>
<td>character or floating point</td>
</tr>
<tr>
<td>evaluation by logical operators</td>
<td>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 internally.</td>
<td>is a known value that, when compared, resolves to a Boolean result</td>
</tr>
<tr>
<td>collating sequence order</td>
<td>appears as the smallest value</td>
<td>appears as the smallest value</td>
</tr>
<tr>
<td>representation in CAS output</td>
<td>a missing numeric value appears as a single period (.). A missing character value appears as a blank space.</td>
<td>a missing numeric value appears as a single period (.). A missing character value appears as a blank space.</td>
</tr>
</tbody>
</table>
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 67.
<table>
<thead>
<tr>
<th>A</th>
<th>BINARY</th>
<th>COMMENT</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABORT</td>
<td>BIT</td>
<td>COMMIT</td>
<td>DATABASE</td>
</tr>
<tr>
<td>ABSOLUTE</td>
<td>BLOB</td>
<td>COMMITTED</td>
<td>DAY</td>
</tr>
<tr>
<td>ACCESS</td>
<td>BOOLEAN</td>
<td>CONDITION</td>
<td>DEALLOCATE</td>
</tr>
<tr>
<td>ACTION</td>
<td>BOTH</td>
<td>CONNECT</td>
<td>DEC</td>
</tr>
<tr>
<td>ADD</td>
<td>BY</td>
<td>CONSTRAINT</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>AFTER</td>
<td>c</td>
<td>CONSTRAINTS</td>
<td>DECLARE</td>
</tr>
<tr>
<td>AGGREGATE</td>
<td>CACHE</td>
<td>CONVERSION</td>
<td>DEFERRED</td>
</tr>
<tr>
<td>ALL</td>
<td>CALL</td>
<td>CONVERT</td>
<td>DEFERRABLE</td>
</tr>
<tr>
<td>ALLOCATE</td>
<td>CALLED</td>
<td>COPY</td>
<td>DEFERRED</td>
</tr>
<tr>
<td>ALTER</td>
<td>CARDINALITY</td>
<td>CORR</td>
<td>DEFINER</td>
</tr>
<tr>
<td>ANALYSE</td>
<td>CASCADE</td>
<td>CORRESPONDING</td>
<td>DELETE</td>
</tr>
<tr>
<td>ANALYZE</td>
<td>CASCADED</td>
<td>COVAR_POP</td>
<td>DELIMITER</td>
</tr>
<tr>
<td>AND</td>
<td>CASE</td>
<td>COVAR_SAMP</td>
<td>DELIMITERS</td>
</tr>
<tr>
<td>ANY</td>
<td>CAST</td>
<td>CREATE</td>
<td>DENSE_RANK</td>
</tr>
<tr>
<td>ARE</td>
<td>CHAIN</td>
<td>CREATEDB</td>
<td>DEREF</td>
</tr>
<tr>
<td>ARRAY</td>
<td>CHAR</td>
<td>CREATEUSER</td>
<td>DESC</td>
</tr>
<tr>
<td>AS</td>
<td>CHAR_LENGTH</td>
<td>CUBE</td>
<td>DESCR</td>
</tr>
<tr>
<td>ASC</td>
<td>CHARACTER</td>
<td>CUME_DIST</td>
<td>DETERMINISTIC</td>
</tr>
<tr>
<td>ASENSITIVE</td>
<td>CHARACTER_LENGTH</td>
<td>CURRENT</td>
<td>DISCONNECT</td>
</tr>
<tr>
<td>ASSERTION</td>
<td>CHARACTERISTICS</td>
<td>CURRENT_DATE</td>
<td>DISTINCT</td>
</tr>
<tr>
<td>ASSIGNMENT</td>
<td>CHECK</td>
<td>CURRENT_TIME</td>
<td>DO</td>
</tr>
<tr>
<td>ASYMMETRIC</td>
<td>CHECKPOINT</td>
<td>CURRENT_TIMESTAMP</td>
<td>DOMAIN</td>
</tr>
<tr>
<td>AT</td>
<td>CLASS</td>
<td>CURRENT_USER</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>ATOMIC</td>
<td>CLOB</td>
<td>CURSOR</td>
<td>DROP</td>
</tr>
<tr>
<td>AUTORIZATION</td>
<td>CLOSE</td>
<td>CYCLE</td>
<td>DYNAMIC</td>
</tr>
<tr>
<td>B</td>
<td>CLUSTER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BACKWARD</td>
<td>COALESCE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEFORE</td>
<td>COLLATE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEGIN</td>
<td>COLLECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BETWEEN</td>
<td>COLUMN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIGINT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 5.3 FedSQL Reserved Words E - O

<table>
<thead>
<tr>
<th>E</th>
<th>G</th>
<th>J</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACH</td>
<td>GET</td>
<td>JOIN</td>
<td>NAMES</td>
</tr>
<tr>
<td>ELEMENT</td>
<td>GLOBAL</td>
<td>K</td>
<td>NATIONAL</td>
</tr>
<tr>
<td>ELSE</td>
<td>GRANT</td>
<td>KEY</td>
<td>NATURAL</td>
</tr>
<tr>
<td>ENCODING</td>
<td>GROUP</td>
<td>L</td>
<td>NCHAR</td>
</tr>
<tr>
<td>ENCRYPTED</td>
<td>GROUPING</td>
<td>LANCOMPILER</td>
<td>NCLOB</td>
</tr>
<tr>
<td>END</td>
<td>H</td>
<td>LANGUAGE</td>
<td>NEW</td>
</tr>
<tr>
<td>END-EXEC</td>
<td>HANDLER</td>
<td>LARGE</td>
<td>NEXT</td>
</tr>
<tr>
<td>ESCAPE</td>
<td>HAVING</td>
<td>LAST</td>
<td>NO</td>
</tr>
<tr>
<td>EVERY</td>
<td>HOLD</td>
<td>LATERAL</td>
<td>NOCREATEDB</td>
</tr>
<tr>
<td>EXCEPT</td>
<td>I</td>
<td>LEADING</td>
<td>NOCREATEUSER</td>
</tr>
<tr>
<td>EXCLUDING</td>
<td>ILIKE</td>
<td>LEFT</td>
<td>NONE</td>
</tr>
<tr>
<td>EXCLUSIVE</td>
<td>IMMEDIATE</td>
<td>LEVEL</td>
<td>NORMALIZE</td>
</tr>
<tr>
<td>EXEC</td>
<td>IMMUNE</td>
<td>LIKE</td>
<td>NOT</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>IMMUTABLE</td>
<td>LIMIT</td>
<td>NOTHING</td>
</tr>
<tr>
<td>EXISTS</td>
<td>IMPLICIT</td>
<td>LISTEN</td>
<td>NOTIFY</td>
</tr>
<tr>
<td>EXPLAIN</td>
<td>IN</td>
<td>LOAD</td>
<td>NOTNULL</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>INCLUDING</td>
<td>LOCAL</td>
<td>NULL</td>
</tr>
<tr>
<td>EXTRACT</td>
<td>INCREMENT</td>
<td>LOCALTIME</td>
<td>NULLIF</td>
</tr>
<tr>
<td>F</td>
<td>INDEX</td>
<td>LOCALTIME</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>FALSE</td>
<td>INDICATOR</td>
<td>LOCATION</td>
<td>O</td>
</tr>
<tr>
<td>FETCH</td>
<td>INHERITS</td>
<td>LOCK</td>
<td>OF</td>
</tr>
<tr>
<td>FILTER</td>
<td>INITIAL</td>
<td>LOAD</td>
<td>OFFSET</td>
</tr>
<tr>
<td>FIRST</td>
<td>INNER</td>
<td>LOCAL</td>
<td>OIDS</td>
</tr>
<tr>
<td>FLOAT</td>
<td>INOUT</td>
<td>LOCALTIME</td>
<td>OLD</td>
</tr>
<tr>
<td>FOR</td>
<td>INPUT</td>
<td>LOCALTIMESTAMP</td>
<td>ON</td>
</tr>
<tr>
<td>FORCE</td>
<td>INSENSITIVE</td>
<td>LOCATION</td>
<td>ONLY</td>
</tr>
<tr>
<td>FOREIGN</td>
<td>INSERT</td>
<td>LOCK</td>
<td>OPEN</td>
</tr>
<tr>
<td>FORWARD</td>
<td>INSTEAD</td>
<td>MATCH</td>
<td>OPERATOR</td>
</tr>
<tr>
<td>FREE</td>
<td>INT</td>
<td>MAXVALUE</td>
<td>OPTION</td>
</tr>
<tr>
<td>FREEZE</td>
<td>INTEGER</td>
<td>MEMBER</td>
<td>OR</td>
</tr>
<tr>
<td>FROM</td>
<td>INTERSECT</td>
<td>MERGE</td>
<td>ORDER</td>
</tr>
<tr>
<td>FULL</td>
<td>INTERSECTION</td>
<td>METHOD</td>
<td>OUT</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>INTERVAL</td>
<td>MIN</td>
<td>OUTER</td>
</tr>
<tr>
<td>FUSION</td>
<td>INTO</td>
<td>MINVALUE</td>
<td>OVER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODE</td>
<td>OVERLAPS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODIFIES</td>
<td>OVERLAY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODULE</td>
<td>OWNER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MONTH</td>
<td></td>
</tr>
</tbody>
</table>
Table 5.4  FedSQL Reserved Words P - Z

<table>
<thead>
<tr>
<th>p</th>
<th>replace</th>
<th>statement</th>
<th>union</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARAMETER</td>
<td>reset</td>
<td>static</td>
<td>unique</td>
</tr>
<tr>
<td>PARTIAL</td>
<td>restart</td>
<td>statistics</td>
<td>unknown</td>
</tr>
<tr>
<td>PARTITION</td>
<td>restrict</td>
<td>stddev_pop</td>
<td>unlisten</td>
</tr>
<tr>
<td>PASSWORD</td>
<td>result</td>
<td>stddev_samp</td>
<td>unnest</td>
</tr>
<tr>
<td>PATH</td>
<td>return</td>
<td>stdin</td>
<td>until</td>
</tr>
<tr>
<td>PENDANT</td>
<td>returns</td>
<td>stdout</td>
<td>update</td>
</tr>
<tr>
<td>PERCENT_RANK</td>
<td>revoke</td>
<td>storage</td>
<td>usage</td>
</tr>
<tr>
<td>PERCENTILE_CONT</td>
<td>right</td>
<td>strict</td>
<td>user</td>
</tr>
<tr>
<td>PERCENTILE_DESC</td>
<td>rollback</td>
<td>submultiset</td>
<td>using</td>
</tr>
<tr>
<td>PLACING</td>
<td>rollback</td>
<td>substring</td>
<td>v</td>
</tr>
<tr>
<td>POSITION</td>
<td>row</td>
<td>symmetric</td>
<td>vacuum</td>
</tr>
<tr>
<td>PRECISION</td>
<td>rows</td>
<td>sysid</td>
<td>valid</td>
</tr>
<tr>
<td>PREPARE</td>
<td>row_number</td>
<td>system</td>
<td>validator</td>
</tr>
<tr>
<td>PRESERVE</td>
<td>rule</td>
<td>system_user</td>
<td>value</td>
</tr>
<tr>
<td>PRIMARY</td>
<td>$</td>
<td>table</td>
<td>values</td>
</tr>
<tr>
<td>PRIOR</td>
<td>schema</td>
<td>tablesample</td>
<td>varchar</td>
</tr>
<tr>
<td>PRIVILEGES</td>
<td>scope</td>
<td>temp</td>
<td>varying</td>
</tr>
<tr>
<td>PROCEDURAL</td>
<td>scroll</td>
<td>template</td>
<td>var_pop</td>
</tr>
<tr>
<td>PROCEDURE</td>
<td>search</td>
<td>temporary</td>
<td>var_samp</td>
</tr>
<tr>
<td>R</td>
<td>secondary</td>
<td>temporal</td>
<td>verbose</td>
</tr>
<tr>
<td>RANK</td>
<td>security</td>
<td>then</td>
<td>version</td>
</tr>
<tr>
<td>READ</td>
<td>select</td>
<td>time</td>
<td>view</td>
</tr>
<tr>
<td>READS</td>
<td>sensitive</td>
<td>timestamp</td>
<td>volatile</td>
</tr>
<tr>
<td>REAL</td>
<td>specific</td>
<td>timezone_hour</td>
<td>when</td>
</tr>
<tr>
<td>RECHECK</td>
<td>specific_type</td>
<td>timezone_minute</td>
<td>whenever</td>
</tr>
<tr>
<td>RECURSIVE</td>
<td>sequence</td>
<td>to</td>
<td>wherever</td>
</tr>
<tr>
<td>REF</td>
<td>serializable</td>
<td>toast</td>
<td>where</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>session</td>
<td>trailing</td>
<td>width_bucket</td>
</tr>
<tr>
<td>REFERENCING</td>
<td>session_user</td>
<td>transaction</td>
<td>window</td>
</tr>
<tr>
<td>REGR_AVGX</td>
<td>set</td>
<td>translate</td>
<td>with</td>
</tr>
<tr>
<td>REGR_AVGY</td>
<td>setof</td>
<td>translation</td>
<td>within</td>
</tr>
<tr>
<td>REGR_COUNT</td>
<td>share</td>
<td>treat</td>
<td>without</td>
</tr>
<tr>
<td>REGR_INTERCEPT</td>
<td>show</td>
<td>trigger</td>
<td>work</td>
</tr>
<tr>
<td>REGR_R2</td>
<td>similar</td>
<td>trim</td>
<td>write</td>
</tr>
<tr>
<td>REGR_SLOPE</td>
<td>simple</td>
<td>true</td>
<td>year</td>
</tr>
<tr>
<td>REGR_SXX</td>
<td>smallint</td>
<td>truncate</td>
<td>z</td>
</tr>
<tr>
<td>REGR_SXY</td>
<td>some</td>
<td>trusted</td>
<td></td>
</tr>
<tr>
<td>REGR_SYY</td>
<td>sqlexception</td>
<td>type</td>
<td></td>
</tr>
<tr>
<td>REINDEX</td>
<td>sqlstate</td>
<td>uescape</td>
<td></td>
</tr>
<tr>
<td>RELATIVE</td>
<td>sqlwarning</td>
<td>values</td>
<td></td>
</tr>
<tr>
<td>YEAR</td>
<td>z</td>
<td>zone</td>
<td></td>
</tr>
<tr>
<td>ZONE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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. A caslib can include in-memory tables or point to files in a caslib’s external data source. caslibs are used to reference libraries in CAS. This is similar to the way librefs identify SAS libraries in SAS.

To show available caslibs:

- From SAS, use the CASLIB statement with the _ALL_ and LIST options. For example:

  ```
cas _all_ list;
  ```

  The preceding code lists all of the caslibs that are available to the current CAS session. For more information, see “CASLIB Statement” in SAS Cloud Analytic Services: User’s Guide.

- When programming with actions, use the table.caslibInfo action. For example:

  ```
table.caslibInfo;
  ```

  By default, the caslibInfo action gets information about all available caslibs. For more information, see “Tables Action Set” in SAS Viya: System Programming Guide.

To list tables that are available from a caslib:

- From SAS, use either of the following:

  - PROC CASUTIL with the LIST statement and the TABLES or FILES parameter. For example:

    ```
    proc casutil;
    list tables;
    list files;
    run;
    ```

    The LIST TABLES statement lists the in-memory tables in the active caslib.
    The LIST FILES statement lists the files in the active caslib. For more information, see “LIST Statement” in SAS Cloud Analytic Services: User’s Guide.

  - PROC DATASETS with a CAS LibNAME engine libref. For example:

    ```
    libname mycas cas caslib=casuser;
    proc datasets lib=mycas;
    run;
    ```

    The libref mycas uses the CAS engine to enable you to access CAS tables in the specified caslib. PROC DATASETS lists the CAS tables in the caslib that is pointed to by libref mycas in the same way that it lists SAS data sets in a SAS library.
With action programming, use the table.tableInfo action. For example:

```sql
  table.tableInfo;
```

By default, the table.tableInfo action lists tables that are available in the active CAS session. For more information, see “Tables Action Set” in SAS Viya: System Programming Guide.

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

- From SAS, use either of the following:
  - PROC CASUTIL with the CONTENTS statement. For example:
    ```sql
    proc casutil;
      contents casdata="table-name";
    run;
    ```
    The CONTENTS statement displays table metadata such as column names and data types for the files or in-memory tables in the active caslib. You must specify an IMPORTOPTIONS=(FILETYPE="file-type") parameter when reading files. For more information, see “CONTENTS Statement” in SAS Cloud Analytic Services: User’s Guide.
  - PROC CONTENTS with a CAS LIBNAME engine libref. For example:
    ```sql
    libname mycas cas caslib=casuser;
    proc contents data=mycas.table-name;
    run;
    ```
    The libref mycas uses the CAS engine to enable you to access CAS tables in the specified caslib. PROC CONTENTS lists column information and details about the specified CAS table the same way that it would list column information and details about a SAS data set.

- With action programming, use the table.columnInfo action. For example:

```sql
  table.columnInfo / table="table-name";
```

The table.columnInfo action displays table metadata such as column names and data types for the files or in-memory tables in the active caslib. You must specify an IMPORTOPTIONS=(FILETYPE="file-type") parameter when reading files.
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

\[ \text{expression} \ \text{[NOT]} \ \text{BETWEEN} \ \text{expression} \ \text{AND} \ \text{expression} \]

Arguments

\text{expression}

specifies any valid SQL expression.

See “<sql-expression> Expression” on page 94

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> Expression” on page 94

CASE Expression

Selects result values that satisfy search conditions and value comparisons.

Valid in: CAS

Syntax

```
\text{CASE} \ [\text{case-expression}]
  \text{WHEN} \ \text{when-expression} \ \text{THEN} \ \text{result-expression}
...
[\text{WHEN} \ \text{when-expression} \ \text{THEN} \ \text{result-expression}]
```
[ELSE result-expression]
END

Arguments

case-expression
specifies any valid SQL expression that evaluates to a table column whose values are compared to when-expression.

See "<sql-expression> Expression" on page 94

Chapter 4, "FedSQL Expressions and Subqueries," on page 51

when-expression
specifies any valid SQL search condition expression or a value expression.

- When case-expression is not specified, when-expression is a search condition expression that evaluates to true or false.
- When case-expression is specified, when-expression is an SQL value expression that is compared to case-expression and that evaluates to true or false.

See "<sql-expression> Expression" on page 94

result-expression
specifies an SQL expression that evaluates to a value.

See "<sql-expression> Expression" on page 94

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:

```sql
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:

```sql
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 155

```sql
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:
Example 2: The CASE Expression Using a Value

Table: WORLDTEMPS on page 155

```sql
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:
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])
```

See Also

Expression:

- “COALESCE Expression” on page 80
- “NULLIF Expression” on page 93
- `<search-condition>` in “SELECT Statement” on page 116
Arguments

expression

specifies any valid SQL expression.

See "<sql-expression> Expression" on page 94

Chapter 4, “FedSQL Expressions and Subqueries,” on page 51

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 “Example: Natural Join” on page 41 and “Example: Natural Join with a WHERE Clause” on page 42.

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 76

DISTINCT Predicate

Specifies that only unique rows can be selected as input or output.

Valid in: CAS
Syntax

Form 1:  \( \text{aggregate-function} (\text{DISTINCT expression}) \);
Form 2:  \textbf{SELECT DISTINCT} <select-list> FROM <table-expression>;

Arguments

\textit{aggregate-function}  
\hspace{10pt} can be any aggregate function.

\textit{expression}  
\hspace{10pt} specifies any valid SQL expression.

See  "<sql-expression> Expression" on page 94

Chapter 4, "FedSQL Expressions and Subqueries," on page 51

\textbf{SELECT <select-list> FROM <table-expression>}
\hspace{10pt} 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 119.

Details

Both forms of the DISTINCT predicate perform a form of duplicate elimination. Use of the DISTINCT keyword with an aggregate function causes FedSQL to only provide unique values as input to the aggregate calculation. Use of SELECT DISTINCT causes FedSQL to only return unique rows in the query’s result set.

Note: Two null values are not considered distinct.

Example

- select count(distinct avghigh) from worldtemps;
- select distinct c1.employee, firstname, salary
  from company as c1;

See Also

Statements:
- “SELECT Statement” on page 116

Functions:
- “AVG Function” in SAS FedSQL Language Reference
- “COUNT Function” in SAS FedSQL Language Reference
- “CSS Function” in SAS FedSQL Language Reference
- “KURTOSIS Function” in SAS FedSQL Language Reference
- “MAX Function” in SAS FedSQL Language Reference
EXISTS Predicate

Tests whether a subquery returns one or more rows.

Valid in: CAS

Syntax

\[[NOT] \text{EXISTS} \ (\text{select-statement})\]

Arguments

\textit{select-statement} specifies a subquery with the SELECT statement.

See \textit{“SELECT Statement” on page 116}

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.

\begin{verbatim}
select *
  from payroll p
  where exists (select * from staff s

EXISTS Predicate
See Also

**Statements:**

- “SELECT Statement” on page 116

---

**IN Predicate**

Tests set membership.

Valid in: CAS

**Syntax**

\[ expression \ [\text{NOT}] \text{ IN ( constant [ , ...constant])} \]

**Arguments**

- **expression**
  
  specifies any valid SQL expression.

- **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 155

```
SELECT city, country
FROM worldtemps
WHERE avghigh IN (90, 97);
```

SAS creates the following table:
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> Expression" on page 94
Chapter 4, “FedSQL Expressions and Subqueries,” on page 51

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: WORLD_CITYCOORDS on page 154
select city
  from worldcitycoords
  where (latitude = 40) is false;

SAS creates the following table:

<table>
<thead>
<tr>
<th>city</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algiers</td>
<td>Algeria</td>
</tr>
<tr>
<td>Calcutta</td>
<td>India</td>
</tr>
<tr>
<td>Lagos</td>
<td>Nigeria</td>
</tr>
</tbody>
</table>
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> Expression" on page 94
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 154

```sql
select *
from worldcitycoords
where city is missing;
```

SAS creates the following table:

<table>
<thead>
<tr>
<th>city</th>
<th>country</th>
<th>latitude</th>
<th>longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>China</td>
<td></td>
<td>40</td>
<td>116</td>
</tr>
</tbody>
</table>

See Also

Predicates:

- "IS NULL Predicate" on page 87
- `<search-condition>` in the "SELECT Statement" on page 116

### 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> Expression” on page 94

Chapter 4, “FedSQL Expressions and Subqueries,” on page 51

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: WORLD_CITYCOORDS on page 154

select city
  from world_citycoords
    where latitude is not null;

SAS creates the following table:
IS TRUE Predicate

Tests for a true value.

Valid in: CAS

Syntax

\( (\text{expression}) \ 	ext{IS} [\text{NOT}] \ TRUE \)

Arguments

\textit{expression}

specifies any valid SQL expression.
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 154
select city
from worldcitycoords
where (latitude = 40) is true;

SAS creates the following table:

Output 6.7  IS TRUE Example Output

See Also

Predicates:
- “IS FALSE Predicate” on page 85
- “IS UNKNOWN Predicate” on page 90

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> Expression" on page 94

Chapter 4, “FedSQL Expressions and Subqueries,” on page 51

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 85
- “IS TRUE Predicate” on page 89
- <search-condition> in the “SELECT Statement” on page 116

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> Expression" on page 94

Chapter 4, “FedSQL Expressions and Subqueries,” on page 51
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.
- `'S%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 147

```sql
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 68.

Syntax

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

Arguments

- `expression` specifies any valid SQL expression.
- **Data type** All data types are valid.
- **See** "<sql-expression> Expression" on page 94
  
  Chapter 4, “FedSQL Expressions and Subqueries,” on page 51

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.

```sql
update grades
set testscore =
CASE
  when testscore = '-1' then null
```
The following code uses the shorter NULLIF expression.

```sql
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 154

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

See Also

Expressions:

- “CASE Expression” on page 76
- “COALESCE Expression” on page 80

---

### `<sql-expression>` Expression

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

**Valid in:** CAS

**Syntax**

```sql
<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 8, “FedSQL Functions,” on page 107

**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 76 or “COALESCE Expression” on page 80.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>adds</td>
</tr>
<tr>
<td>-</td>
<td>subtracts</td>
</tr>
<tr>
<td>*</td>
<td>multiplies</td>
</tr>
<tr>
<td>/</td>
<td>divides</td>
</tr>
<tr>
<td>=</td>
<td>equals</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>does not equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>is greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>is less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>is greater than or equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>is less than or equal to</td>
</tr>
<tr>
<td>**</td>
<td>raises to a power</td>
</tr>
<tr>
<td>unary –</td>
<td>indicates a negative number</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**predicate**
is an expression that returns true, false, or unknown.

The following predicates are valid.

- “BETWEEN Predicate” on page 75
- “DISTINCT Predicate” on page 81
- “EXISTS Predicate” on page 83
- “IN Predicate” on page 84
- “IS FALSE Predicate” on page 85
- “IS MISSING Predicate” on page 86
- “IS NULL Predicate” on page 87
- “IS TRUE Predicate” on page 89
- “IS UNKNOWN Predicate” on page 90
- “LIKE Predicate” on page 91

**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>
<thead>
<tr>
<th>Group</th>
<th>Expressions, Operators, and Predicates</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( )</td>
<td>forces the expression enclosed to be evaluated first</td>
</tr>
<tr>
<td>1</td>
<td>CASE expression</td>
<td>See “CASE Expression” on page 76</td>
</tr>
<tr>
<td>2</td>
<td>**</td>
<td>raises to a power</td>
</tr>
<tr>
<td></td>
<td>unary +, unary −</td>
<td>indicates a positive or negative number</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>multiplies</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>divides</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>adds</td>
</tr>
<tr>
<td></td>
<td>−</td>
<td>subtracts</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>[NOT] BETWEEN predicate</td>
<td>See “BETWEEN Predicate” on page 75.</td>
</tr>
<tr>
<td></td>
<td>DISTINCT predicate</td>
<td>See “DISTINCT Predicate” on page 81</td>
</tr>
<tr>
<td></td>
<td>[NOT] EXISTS predicate</td>
<td>See “EXISTS Predicate” on page 83</td>
</tr>
<tr>
<td></td>
<td>[NOT] IN predicate</td>
<td>See “IN Predicate” on page 84</td>
</tr>
<tr>
<td></td>
<td>IS [NOT] TRUE predicate</td>
<td>See “IS TRUE Predicate” on page 89</td>
</tr>
<tr>
<td></td>
<td>IS [NOT] FALSE predicate</td>
<td>See “IS FALSE Predicate” on page 85</td>
</tr>
<tr>
<td></td>
<td>IS [NOT] MISSING predicate</td>
<td>See “IS MISSING Predicate” on page 86</td>
</tr>
<tr>
<td></td>
<td>IS [NOT] NULL predicate</td>
<td>See “IS NULL Predicate” on page 87</td>
</tr>
<tr>
<td></td>
<td>IS [NOT] UNKNOWN predicate</td>
<td>See “IS UNKNOWN Predicate” on page 90</td>
</tr>
<tr>
<td></td>
<td>LIKE predicate</td>
<td>See “LIKE Predicate” on page 91</td>
</tr>
</tbody>
</table>
**Expressions, Operators, and Predicates**

<table>
<thead>
<tr>
<th>Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>equals</td>
</tr>
<tr>
<td>^=, &lt;&gt;</td>
<td>does not equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>is greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>is less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>is greater than or equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>is less than or equal to</td>
</tr>
<tr>
<td>8</td>
<td>AND</td>
</tr>
<tr>
<td>9</td>
<td>OR</td>
</tr>
<tr>
<td>10</td>
<td>NOT</td>
</tr>
</tbody>
</table>

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 $A+B\times C$ as $A+(B\times C)$, although you can add parentheses to make it evaluate as $(A+B)\times C$ for a different result.

**See Also**

**Statements:**

- “SELECT Statement” on page 116
- Chapter 4, “FedSQL Expressions and Subqueries,” on page 51
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.


Validation of FedSQL Formats

The PUT function validates the specified format upon use.

FedSQL Format Examples

```sql
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.

```sas
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 7.1 on page 101. The output of the PUT function is shown in Figure 7.2 on page 101.

Figure 7.1 Content of the Source EMPLOYEES Table

<table>
<thead>
<tr>
<th>EMP_NAME</th>
<th>HIRE_DATE</th>
<th>DEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greg Welty</td>
<td>26NOV2004</td>
<td>20</td>
</tr>
<tr>
<td>Penny Jackson</td>
<td>26NOV2004</td>
<td>20</td>
</tr>
<tr>
<td>Edward Murray</td>
<td>26NOV2004</td>
<td>10</td>
</tr>
<tr>
<td>Ronald Thomas</td>
<td>26NOV2004</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 7.2 Content of the Employees Table After the PUT Function Is Applied

<table>
<thead>
<tr>
<th>EMP_NAME</th>
<th>HIRE_DATE</th>
<th>DEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greg Welty</td>
<td>26NOV2004</td>
<td>Research</td>
</tr>
<tr>
<td>Penny Jackson</td>
<td>26NOV2004</td>
<td>Research</td>
</tr>
<tr>
<td>Edward Murray</td>
<td>26NOV2004</td>
<td>Sales</td>
</tr>
<tr>
<td>Ronald Thomas</td>
<td>26NOV2004</td>
<td>Sales</td>
</tr>
</tbody>
</table>

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*.

<table>
<thead>
<tr>
<th>Category</th>
<th>Language Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date and Time</td>
<td>NLDATEmw.</td>
<td>Converts a SAS date value to the date value of the specified locale, and then writes the date value as a date.</td>
</tr>
<tr>
<td></td>
<td>NLDATEMDmw.</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>NLDATEMNmw.</td>
<td>Converts a SAS date value to the date value of the specified locale, and then writes the value as the name of the month.</td>
</tr>
<tr>
<td></td>
<td>NLDATEmw.</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>NLDATENw.</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>NLDATEmw</td>
<td>Converts the SAS date value to the date value of the specified locale, and then writes the value as the year and the name of the month.</td>
</tr>
<tr>
<td></td>
<td>NLDATEYWw.</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>NLDATEYRw.</td>
<td>Converts the SAS date value to the date value of the specified locale, and then writes the date value as the year.</td>
</tr>
<tr>
<td></td>
<td>NLDATEYWw.</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>NLDATMAPw.</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>NLDATMDTuw.</td>
<td>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.</td>
</tr>
<tr>
<td>Category</td>
<td>Language Element</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>NLDATMMDw.</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>NLDATMMNw.</td>
<td>Converts the SAS datetime value to the datetime value of the specified locale, and then writes the value as the name of the month.</td>
</tr>
<tr>
<td></td>
<td>NLDATMTMw.</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>NLDATMw.</td>
<td>Converts a SAS datetime value to the datetime value of the specified locale, and then writes the value as a datetime.</td>
</tr>
<tr>
<td></td>
<td>NLDATMWw.</td>
<td>Converts a SAS datetime value to the day of the week, date, and time of the specified locale.</td>
</tr>
<tr>
<td></td>
<td>NLDATMWNw.</td>
<td>Converts a SAS datetime value to the day of the week of the specified locale.</td>
</tr>
<tr>
<td></td>
<td>NLDATMYMw.</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>NLDATMYQw.</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>NLDATMYRw.</td>
<td>Converts the SAS datetime value to the datetime value of the specified locale, and then writes the value as the year.</td>
</tr>
<tr>
<td></td>
<td>NLDATMYWw.</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>NLTIMAPw.</td>
<td>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.</td>
</tr>
<tr>
<td>Category</td>
<td>Language Element</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>NLTIMEw.</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Numeric</td>
<td>NLBESTw.</td>
<td>Writes the best numerical notation based on the locale.</td>
</tr>
<tr>
<td></td>
<td>NLMNYw.d</td>
<td>Writes the monetary format of the local expression in the specified locale using local currency.</td>
</tr>
<tr>
<td></td>
<td>NLMNYIw.d</td>
<td>Writes the monetary format of the international expression in the specified locale.</td>
</tr>
<tr>
<td></td>
<td>NLNUMw.d</td>
<td>Writes the numeric format of the local expression in the specified locale.</td>
</tr>
<tr>
<td></td>
<td>NLNUMIw.d</td>
<td>Writes the numeric format of the international expression in the specified locale.</td>
</tr>
<tr>
<td></td>
<td>NLPCTw.d</td>
<td>Writes percentage data of the local expression in the specified locale.</td>
</tr>
<tr>
<td></td>
<td>NLPCTIw.d</td>
<td>Writes percentage data of the international expression in the specified locale.</td>
</tr>
<tr>
<td></td>
<td>NLPCTNw.d</td>
<td>Produces percentages, using a minus sign for negative values.</td>
</tr>
<tr>
<td></td>
<td>NLPCTPw.d</td>
<td>Writes locale-specific numeric values as percentages. Writes locale-specific numeric values as percentages.</td>
</tr>
<tr>
<td></td>
<td>NLPVALUEw.d</td>
<td>Writes p-values of the local expression in the specified locale.</td>
</tr>
<tr>
<td></td>
<td>NLSTRMONw.d</td>
<td>Writes the month name in the specified locale.</td>
</tr>
<tr>
<td></td>
<td>NLSTRQTRw.d</td>
<td>Writes a numeric value as the quarter-of-the-year in the specified locale.</td>
</tr>
<tr>
<td></td>
<td>NLSTRWKw.d</td>
<td>Writes a numeric value as the day-of-the-week in the specified locale.</td>
</tr>
</tbody>
</table>
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.

Beginning in SAS Viya 3.5, the CAST function is supported in CAS. The CAST function enables you to convert a value from one data type to another, or from one length to another.

When using FedSQL functions, note these points:

- Within the functions, the SQL expressions in function arguments are limited to the SQL expressions that are supported in CAS. For more information, see "<sql-expression> Expression" on page 94.
- 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 and DATE, TIME, and TIMESTAMP 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:

```plaintext
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.

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

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

```plaintext
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:

```plaintext
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, time, and datetime values.
A SAS date value is the number of days between January 1, 1960, and a specified date. Dates before January 1, 1960, are negative numbers; dates after are positive numbers. For example, the SAS date value for January 1, 1960, is 0, -365 for January 1, 1959, and 17532 for January 1, 2008.

A SAS time value is the number of seconds since midnight of the current day. SAS time values are between 0 and 86400.

A SAS datetime value is the number of seconds between January 1, 1960, and a specific hour, minute, and second of a specific date.

When used in an expression, the output of these functions is unintelligible unless you use a function to format the values. Consider the following request that returns the current date with the TODAY function:

```plaintext
proc cas;
    fedsql.execdirect query="select today();
run;
```

Output 8.1 Default Output of the TODAY Function

<table>
<thead>
<tr>
<th>TODAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>21802</td>
</tr>
</tbody>
</table>

There are two ways to format date, time, and datetime output values:

- by using the PUT function. The PUT function enables you to apply a SAS format to the SAS output value.
- beginning with SAS Viya 3.5, by using the CAST function. The CAST function converts the DOUBLE column containing the values to an ANSI DATE, TIME, or TIMESTAMP type.

The following example shows how to format the output of the TODAY function with the PUT function:

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

Output 8.2 Output of the TODAY Function When Formatted with the PUT Function

<table>
<thead>
<tr>
<th>column</th>
</tr>
</thead>
<tbody>
<tr>
<td>10SEP19</td>
</tr>
</tbody>
</table>

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. A column that is formatted with the PUT function is treated like a character column elsewhere in the request. In addition, it does not have a title unless you specify a column alias. For information about the PUT function, see "PUT Function" in SAS FedSQL Language Reference.

The following example shows how to format the output of the TODAY function with the CAST function:

```plaintext
proc cas;
    fedsql.execdirect query="select today():: date; run;
```
The CAST function displays the value in a readable format without your having to specify a SAS format. The name of the TODAY function is retained as the column heading. In addition, if the table containing the column is later to be saved to an ANSI-compatible DBMS, the column would be in the expected type for the DBMS. For more information about the CAST function, see “CAST 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 Statements

Dictionary

CREATE TABLE Statement .................................................. 113
DROP TABLE Statement .................................................... 115
SELECT Statement .......................................................... 116

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 an existing table by using the DROP TABLE statement before creating a table of the same name. Or, you can specify the REPLACE= table option in the CREATE TABLE statement. See “REPLACE= Table Option” on page 134.

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.
The OPTIONS argument and all table options must be enclosed in braces ({}).

See “FedSQL Table Options” on page 131

**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 114

Chapter 4, “FedSQL Expressions and Subqueries,” on page 51

“SELECT Statement” on page 116

**Details**

Overview of the CREATE TABLE Statement for CAS

The CREATE TABLE statement for CAS enables you to create a table by selecting columns from one or more existing tables using a query expression. The FedSQL language supports the creation of CAS output tables from CAS input tables and DBMS input tables. The new tables are session tables. You must use another action to save or promote the tables.

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.

For example, this CREATE TABLE statement creates a new table that is based on only three columns from the CorpData table:

```sql
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:

```sql
create table spain
as select * from corpdata where country='Spain';
```

For information about other query approaches, see Chapter 4, “FedSQL Expressions and Subqueries,” on page 51.

The output table preserves any formats that were defined on the input tables. FedSQL does not preserve table labels from input tables. Use the LABEL= table option to assign a label to an output table.
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 the active caslib, use a two-part name in the form `caslib.table-name`.

- **FORCE**
  specifies that the table is dropped without error processing.

Details

By default, FedSQL will not overwrite an existing in-memory table of the same name. If you want to create a replacement session table for an existing session table of the same name, you must either first drop the existing table from your CAS session by using the DROP TABLE statement. Or, you can specify the REPLACE= table option in your CREATE TABLE request. For more information about this table option, see "REPLACE= Table Option" on page 134..

The DROP TABLE statement writes messages to the log when the specified table cannot be found. These messages can interfere with a CREATE TABLE statement that is included in the same request as the DROP TABLE statement. The FORCE option suppresses the messages so that the CREATE TABLE request can succeed. Another way to stop the interference is to specify NOERRORSTOP in the PROC FEDSQL statement.

The DROP TABLE statement removes an in-memory table from your CAS session. It does not affect CAS tables that have been promoted for global use. Nor can the DROP TABLE statement remove a CAS table that is saved to disk.
CAS session tables that are created with FedSQL exist for the duration of the CAS session, unless you save or promote the tables with another action.

Examples:

Example 1: Drop a Table

```sql
proc fedsql sessref=casauto;
    drop table mytable;
quit;
```

Example 2: Drop a Table and Create a Table at the Same Time

```sql
proc fedsql sessref=casauto;
    drop table mytable force;
    create table mytable as
    select a, b, c from bigtable
    where country="United States";
quit;
```

See Also

Table Options:
- "REPLACE= Table Option" on page 134

### SELECT Statement

Retrieves columns and rows of data from tables.

**Valid in:** CAS  
**Categories:** Data Definition, Data Manipulation  
**Note:** VARBINARY columns are processed by the SELECT statement; however, the columns might not be visible in some clients. In SAS clients, you can apply a format to the columns with the PUT function, and the columns are displayed in the specified format.

**Syntax**

The main clauses of the SELECT statement can be summarized as follows.

```sql
SELECT <select-list>
    FROM <table-specification>
    [WHERE <search-condition>]
    [GROUP BY <grouping-column>]
    [HAVING <search-condition>]
    [ORDER BY <sort-specification>]
    [LIMIT {count | ALL}]
```
The detailed syntax of the SELECT statement is as follows.

```
<query-expression>
    [ORDER BY <sort-specification> [ , ...<sort-specification> ]];
<query-expression>::=
    {<query-specification> | <query-expression>}
    {UNION [ALL] [CORRESPONDING [BY (column, column, ...)]]}<query-specification>
<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]
    | CONNECTION TO catalog (<native-syntax>) [AS alias]
    | (<query-specification>) [AS alias]
    | <joined-table>
<joined-table>::=
    <cross-join>
    | <qualified-join>
    | <natural-join>
    <cross-join>::=
    <table-specification> CROSS JOIN <table-specification>
<qualified-join>::=
    <table-specification> [<join-type>] JOIN <table-specification> <join-specification>
<natural-join>::=
    <table-specification> NATURAL [<join-type>] JOIN <table-specification>
<join-type>::=
    INNER
    | LEFT [OUTER]
    | RIGHT [OUTER]
    | FULL [OUTER]
<join-specification>::=
```
ON <search-condition>
   | USING (column [, ...column])

<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 119
- “FROM Clause” on page 120
- “WHERE Clause” on page 122
- “GROUP BY Clause” on page 123
- “HAVING Clause” on page 124
- “ORDER BY Clause” on page 125
- “LIMIT Clause” on page 126
- “OFFSET Clause” on page 127
- “UNION Operator” on page 127

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 is 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 result set. 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 the virtual result set.

**Syntax**

```
SELECT [ALL | DISTINCT] <select-list>
<select-list>::=
  *
  | column [AS column-alias]
  | <sql-expression> [AS column-alias]
  | table.*
  | table-alias.*
  | <query-specification>
```

**Asterisk (*) Notation**

The asterisk (*) represents all columns of the table or tables that are listed in the FROM clause. When an asterisk is not prefixed with a table name, all the columns from all tables in the FROM clause are included; when it is prefixed (for example, `table.*` or `table-alias.*`), all the columns from only that table are included.

**Column Aliases**

A column alias is a temporary, alternate name for a column. Aliases are specified in the SELECT clause to name or rename columns in the result set in order to be clearer or easier to read. Aliases are often used to name a column that is the result of an arithmetic expression or summary function.

An alias is 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 67. 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
  | CONNECTION TO catalog (<native-syntax>) [AS] alias
  | ( <query-specification>) [AS] alias
  | <joined-table>

<joined-table>::=
  <cross-join>
  | <qualified-join>
  | <natural-join>

<cross-join>::=
  <table-specification> CROSS JOIN <table-specification>

<qualified-join>::=
  <table-specification> [ <join-type>] JOIN <table-specification> <join-specification>

<natural-join>::=
  <table-specification> NATURAL [ <join-type>] JOIN <table-specification>

<join-type>::=
  INNER
  | LEFT [OUTER]
  | RIGHT [OUTER]
  | FULL [OUTER]

<join-specification>::=
  ON <search-condition>
  | USING (column [, …column])

Arguments

CONNECTION TO catalog (<native-syntax>) [AS] alias

specifies data from a DBMS catalog by using the SQL pass-through facility. You can use SQL syntax that the DBMS understands, even if that syntax is not valid in FedSQL.

CROSS JOIN

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

See  "Example: Simple Cross Join" on page 34, "Example: Cross Join with Specified Columns and a WHERE Clause" on page 36
JOIN
defines a join that enables you to filter the data by using a search condition or by using specific columns.

See  "Example: Qualified Join with an ON Clause" on page 37, "Example: Qualified Join with a USING Clause" on page 38, "Example: Qualified Join with an ON Clause and a WHERE Clause" on page 40

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  "Example: Natural Join" on page 41, "Example: Natural Join with a WHERE Clause" on page 42

(<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 56

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

table-name
the name of an in-memory table in the current CAS session.

caslib.table-name
the name of a table that is persisted on the CAS server or exists in an external data source. The caslib points to a library definition for a data source connector. The definition contains data source connection details, such as host, user name, password, and data access specifics, such as path or database, catalog, and schema.

FedSQL requires a standard SQL name to access a data source. It supports two-part names in the form catalog.table-name or schema.table-name and three-part names in the form catalog.schema.table-name. The caslib generates a one- or two-part qualifier for the table name to create an SQL name that is appropriate for the data source and sends it to FedSQL. For more information about caslibs, see “Caslibs” in An Introduction to SAS Viya Programming.

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 non-matching rows from the first and second table
are returned.

column
specifies the name of a column.

ON <search-condition>
specifies a condition join used to match rows from one table to another. If the
search condition is satisfied, the matching rows are added to the result set.

See "<search-condition>" on page 129

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

See "Understanding Inner and Outer Join Types" on page 43

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 result set. The result set contains data from each contributing
table. These queries are referred to as joins. Joins do not alter the original table.

Conceptually, when two tables are specified, each row of table A is matched with all
the rows of table B to produce an internal or intermediate result set. The number of
rows in the intermediate result set (Cartesian) is equal to the product of the number
of rows in each of the source tables. The intermediate result set 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 28.

WHERE Clause
Description
Subsets the result set based on the specified search conditions.

Syntax
WHERE <search-condition>
Arguments

<search-condition>
specifies the conditions for the rows returned by the WHERE clause.

See "<search-condition>" on page 129

Details

The WHERE clause requires a search condition (one or more expressions separated by an operand or predicate) that specifies which rows are chosen for inclusion in the result set. When a condition is met (that is, the condition resolves to true), those rows are displayed in the result set. Otherwise, no rows are displayed.

Note: You cannot use aggregate functions that specify only one column. For example, you cannot use the following code.

where max(inventory1)>10000;

However, you can use this WHERE clause.

where max(inventory1, inventory2)>10000;

Note: If a column contains REAL or DOUBLE values, avoid using a WHERE clause with the = and the <> operators. REAL and DOUBLE values are approximate numeric data types and can give inaccurate results when used in a WHERE clause with the = and the <> operators. You should limit REAL and DOUBLE columns to comparisons with the > or < operator.

GROUP BY Clause

Description

Specifies how to group the data for summarizing.

Syntax

GROUP BY <grouping-column> [ , …<grouping-column>]  

<grouping-column>::=  

   column [ , …column]  
   | column-position-number  
   | <sql-expression>

Arguments

column
specifies the name of a column or a column alias.

column-position-number
specifies a nonnegative integer that equates to a column position.

<sql-expression>
specifies a valid SQL expression.

See "<sql-expression> Expression" on page 94
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 FedSQL in how to summarize the data for each group. When you use a GROUP BY clause without an aggregate function, FedSQL 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.

FedSQL does not support the PROC SQL remerge feature. In FedSQL, the results of an analysis can be integrated with other columns from the data by using a subquery. For more information, see "Rewriting a GROUP BY Remerge Query in FedSQL" in SAS FedSQL Language Reference.

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 129
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 9.1 Differences between the HAVING Clause and WHERE Clause

<table>
<thead>
<tr>
<th>HAVING clause attributes</th>
<th>WHERE clause attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>typically used to specify conditions for including or excluding groups of rows from a table</td>
<td>used to specify conditions for including or excluding individual rows from a table</td>
</tr>
<tr>
<td>must follow the GROUP BY clause in a query, if used with a GROUP BY clause</td>
<td>must precede the GROUP BY clause in a query, if used with a GROUP BY clause</td>
</tr>
<tr>
<td>affected by a GROUP BY clause; when there is no GROUP BY clause, the HAVING clause is treated like a WHERE clause</td>
<td>not affected by a GROUP BY clause</td>
</tr>
<tr>
<td>processed after the GROUP BY clause and any aggregate functions</td>
<td>processed before a GROUP BY clause, if there is one, and before any aggregate functions</td>
</tr>
</tbody>
</table>

ORDER BY Clause

Description

Specifies the order in which rows are returned in the result set.

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> Expression" on page 94

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.

UNION Operator
Descriptions
Combines the result of two SELECT statements into a single result set that contains the unique rows from both queries.

Syntax
{<query-specification> | <query-expression>}

    UNION [ALL][CORRESPONDING [BY (column, ...column)]]
{<query-specification> | <query-expression>}


Arguments

<query-specification> | <query-expression>
specifies a SELECT statement that produces a virtual result set.

See “SELECT Statement” on page 116
“Subqueries” on page 56

UNION
specifies that the result sets are combined and returned as a single result set.

See Chapter 4, “FedSQL Expressions and Subqueries,” on page 51

ALL
(optional) specifies that all rows, including duplicates, are included in the result set.

CORRESPONDING
(optional) specifies to combine columns in the result set by column name.

BY (column, [...column])
specifies that only columns in the column list be included in the result set. Columns are identified by column name.

Restriction The column names must exist in both input tables.

Details

The UNION set operator produces a result set that contains all the unique rows that result from two queries. That is, the result set contains rows that are produced by the first query and the second query. Any subsequent queries that are specified and separated by a UNION operator are compared to the result set of the first two queries; for example, (A UNION B) UNION C. The UNION set operator automatically eliminates duplicate rows from its result sets. The optional ALL keyword is available to preserve the duplicate rows, reduce the execution by one step, and thereby improve the query's performance.

Columns are appended by position in the tables by default, regardless of the column names. However, the data type of the corresponding columns must match or be of a corresponding data type where length is the only differentiator, or the union does not occur. That is, CHAR can be merged with CHAR or VARCHAR, and INTEGER can be merged with BIGINT, INTEGER, SMALLINT, or TINYINT. However, CHAR or INTEGER cannot be merged with DOUBLE.

When columns of different lengths are merged, the length of the larger column is used. For example, if one column is CHAR(4) and the corresponding column is CHAR(10), then CHAR(10) is used.

The optional CORRESPONDING keyword causes FedSQL to match columns by column name instead of by position. The name matching is not case-sensitive. Formats are ignored unless they originate in the first query. Any columns that have non-matching column names are ignored. Use the CORRESPONDING keyword when the columns that you want to match have the same column name and general type but exist in different positions in the input tables.

CORRESPONDING without a column list forms a result set that contains all columns that have the same name. The BY keyword restricts the columns that are returned to those whose names are in the column list. Columns that are not specified in the column list are dropped, even if the original queries specify SELECT
Use the BY keyword to include just one or a few of the columns from both queries in the result set.

The names of the columns in the result set are the names of the columns from the first query expression or query-specification unless a column (such as an expression) has no name in the first query expression or query-specification. Then, the name used depends on whether the SELECT statement result set is being returned to the client, or whether the SELECT statement is part of a CREATE TABLE AS statement. When the SELECT statement result set is being returned to the client, the name of an unnamed column in the result table is "column". Unnamed columns in a CREATE TABLE AS statement are given generated names. The generated names are _expr_1, _expr_2, and so on.

See Also

Chapter 4, “FedSQL Expressions and Subqueries,” on page 51

<search-condition>

Description

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

Syntax

<search-condition>::=

{  
  [NOT] {<sql-expression> | (<search-condition>)}  
  [{AND | OR} [NOT] {<sql-expression> | (<search-condition>)}]  
}  
[, ...{NOT} {<sql-expression> | (<search-condition>)}  
  [{AND | OR} [NOT] {<sql-expression> | (<search-condition>)}]][]

<sql-expression>::=  
  expression {operator | predicate} expression

Arguments

NOT

negates a Boolean condition. This table outlines the outcomes when you compare true and false values using the NOT operator.

Table 9.2  Truth Table for the NOT Operator

<table>
<thead>
<tr>
<th>NOT</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>Unknown</td>
<td>Unknown</td>
</tr>
</tbody>
</table>
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 9.3  Truth Table for the AND Operator

<table>
<thead>
<tr>
<th>AND</th>
<th>True</th>
<th>False</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
<td>Unknown</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Unknown</td>
<td>Unknown</td>
<td>False</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

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 9.4  Truth Table for the OR Operator

<table>
<thead>
<tr>
<th>OR</th>
<th>True</th>
<th>False</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>Unknown</td>
</tr>
<tr>
<td>Unknown</td>
<td>True</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

<sql-expression> specifies any valid SQL expression.

See "<sql-expression> Expression" on page 94

Details

The search condition specifies which rows are returned in a result set for a SELECT statement. Within the SELECT statement, the search condition is used in the WHERE clause, the HAVING clause, or 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.
FedSQL Table Options

Overview of Statement Table Options

About FedSQL Statement Table Options

Restrictions

How to Specify FedSQL Statement Table Options

Dictionary

COMPRESS= Table Option

LABEL= Table Option

REPLACE= Table Option

REPLICATION= Table Option

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

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**

```plaintext
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.

```sql
/* 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.

<table>
<thead>
<tr>
<th>Valid in</th>
<th>CAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category:</td>
<td>Table Control</td>
</tr>
<tr>
<td>Default:</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

**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 of the same name. 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.

REPLACE= removes in-memory session tables only. REPLACE= has no effect on tables that have been promoted for global use. Nor does it affect tables that have been saved to disk.

Example

```sql
create table mytable {options replace=true};
```

See Also

Statements:
- “DROP TABLE Statement” on page 115

---

**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

```sql
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.
Appendixes

Appendix 1
Trouble Shooting: FedSQL Error Codes .................................................. 139

Appendix 2
Tables Used in Examples ................................................................. 143

Appendix 3
ICU License Agreement ................................................................. 157
Resolving FedSQL Errors

Types of Errors

FedSQL writes errors to the SAS log when it is unable to complete a request. Most errors can be categorized as follows:

Usage errors
Usage errors arise because of incorrect or invalid requests. FedSQL usage errors are typically caused by syntax, semantic, or logic errors. Syntax and semantic errors are detected at compile time and prevent compilation of a FedSQL statement. A FedSQL statement that successfully compiles might still contain logic errors that surface at execution time.

Resource errors
Resource errors arise when resource service requests cannot be fulfilled. Various conditions lead to a resource service request failure. For example, the resource could be inaccessible, a resource threshold could be reached, or you could lack authorization to access the resource.

How to Resolve FedSQL Errors

To resolve errors:

1. Check the SAS log for error messages. These messages might help you understand the source of the problem or provide ideas to help you resolve the error.
Here is an example that shows what the problem is. In this instance, a usage error occurred. The messages indicate that a problem occurred in the compilation phase and was either a syntax error or access violation. The problem was encountered at the word DRIP. In this instance, a review of the code showed that the word was encountered where a table option was expected. A quick check of the documentation shows that FedSQL does not support a DRIP table option. It does support a DROP= table option.

ERROR: Compilation error.
ERROR: Syntax error or access violation DRIP

2 In the SAS log, focus on the earliest error messages first. Sometimes a single problem leads to a cascade of failures and resolving the first error can also resolve subsequent errors.

3 If an error message indicates a resource error, check with your system administrator to monitor system resources. Examples of system resources include memory usage and disk space. If system resources are low, look for ways to increase resources or consider ways that your application could use them more efficiently.

Here is an example that shows a resource error. The message indicates a memory threshold was reached.

ERROR: Memory allocation failed.

4 If you have been unable to resolve the error using the previous three steps, contact SAS Technical Support. Here is a list of information you will need to supply: Four tips to remember when you contact SAS Technical Support.

If the SAS log contains any error messages with internal error codes, provide the codes to your SAS Technical Support representative. Internal error codes are hexadecimal numbers that can help SAS more quickly determine the source of the problem. Here is an example of an error message with an internal error code.

ERROR: FedSQL internal error code 0xECCC8CBBF7C049FD. Refer to FedSQL documentation for resolving errors.

---

Additional Information to Provide Technical Support for FedSQL Requests Submitted in a CAS Library

Beginning with SAS Viya 3.5, FedSQL supports a _DIAG option that can be used to gather additional information for SAS Technical Support. Your SAS Technical Support representative will request this information from you when you encounter an internal error. The messages returned by the _DIAG option are intended for FedSQL developers and will not make sense to you.

Add the _DIAG option to the FedSQL request that failed. The following code shows how to add the option to the FedSQL procedure. The _DIAG option is specified as a procedure option.

```sas
proc fedsql sessref=casauto _diag;
...FedSQL statements...
quit;
```
Be sure to specify the SESSREF= or SESSUUID= procedure option in addition to the _DIAG procedure option. In SESSREF= (or SESSUUID=), specify the CAS session in which the problem occurred.

In the fedSql.execDirect action, the _DIAG option is specified as an action parameter. Action parameters are specified without a preceding underscore and take a True or False value. Here is an example:

```sas
proc cas;
  fedsql.execdirect
    diag=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.

Other information that is helpful to provide to SAS Technical Support includes the following:

- the number of tables involved in the failed request
- the number of columns and rows in each input table
- the data types used
- the data sources involved in the request
- the CASLIB statement or addCaslib action used to make the data source connection or connections
- the showStages output for the request. See "Viewing the FedSQL Query Plan" on page 14.
Appendix 2

Tables Used in Examples

Overview of Sample Tables ................................................................. 144
AfewWords ....................................................................................... 144
  Code ......................................................................................... 144
  Content .................................................................................. 144
Customers ....................................................................................... 145
  Code ......................................................................................... 145
  Content .................................................................................. 145
CustonLine ...................................................................................... 146
  Code ......................................................................................... 146
  Content .................................................................................. 147
Densities .......................................................................................... 147
  Code ......................................................................................... 147
  Content .................................................................................. 148
Employees ...................................................................................... 148
  Code ......................................................................................... 148
  Content .................................................................................. 149
Numbers1 ....................................................................................... 149
  Code ......................................................................................... 149
  Content .................................................................................. 150
Numbers2 ....................................................................................... 150
  Code ......................................................................................... 150
  Content .................................................................................. 151
Numbers3 ....................................................................................... 151
  Code ......................................................................................... 151
  Content .................................................................................. 151
Products .......................................................................................... 152
  Code ......................................................................................... 152
  Content .................................................................................. 152
Sales ............................................................................................... 153
  Code ......................................................................................... 153
  Content .................................................................................. 153
WorldCityCoords ........................................................................... 154
  Code ......................................................................................... 154
  Content .................................................................................. 155
WorldTemps ................................................................................... 155
  Code ......................................................................................... 155
  Content .................................................................................. 156
Overview of Sample Tables

This section includes output listings of the tables that are used in the examples in this documentation and in the documentation for the `fedSql.execDirect` action. It also includes the code that is necessary to create the tables in CAS. The FedSQL language INSERT statement is currently not supported in CAS; therefore, SAS DS2 language code is provided to create the tables. You can submit the DS2 code by establishing a session on your CAS server and then using the DS2 procedure or the `ds2.runDS2` action to submit the code. In PROC DS2, be sure to specify the `SESSREF=` procedure option to direct the request to the CAS server. The code creates in-memory CAS tables. You can use the `table.save` action to save the tables to the CAS server. If you have licensed SAS Data Connector software, you can save the tables to a database.

AfewWords

Code

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

```ds2
data afewwords;
  dcl char(10) Word1;
  dcl char(10) Word2;
  method run();
    Word1='*some/'; Word2='WHERE'; output;
    Word1='*every*'; Word2='THING'; output;
    Word1='*no*'; Word2='BODY'; output;
  end;
enddata;
run;
```

Content

<table>
<thead>
<tr>
<th>Word1</th>
<th>Word2</th>
</tr>
</thead>
<tbody>
<tr>
<td>*some/</td>
<td>WHERE</td>
</tr>
<tr>
<td><em>every</em></td>
<td>THING</td>
</tr>
<tr>
<td><em>no</em></td>
<td>BODY</td>
</tr>
</tbody>
</table>
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;

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

Code

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

data custonline;
  dcl varchar(15) custnum having label 'Customer Number';
  dcl timestamp begintime having label 'Begin Time';
  dcl timestamp endtime having label 'End Time';
  method run();
    custnum='US-C-37533944'; begintime=timestamp'2013-09-01 10:00:00';
    endtime=timestamp'2013-09-01 10:05:01.253'; output;
    custnum='GB-W-33944332'; begintime=timestamp'2013-10-02 22:15:33';
    endtime=timestamp'2013-10-02 22:21:09.421'; output;
    custnum='SP-M-29443992'; begintime=timestamp'2013-10-15 18:44:25';
    endtime=timestamp'2013-10-15 19:04:55.746'; output;
    endtime=timestamp'2013-11-01 12:25:09.398'; output;
    custnum='FR-P-98384488'; begintime=timestamp'2013-12-01 12:15:34';
    endtime=timestamp'2013-12-01 12:47:45.221'; output;
    custnum='GB-L-24995559'; begintime=timestamp'2013-01-02 15:43:24';
    endtime=timestamp'2013-01-02 16:06:15.766'; output;
    custnum='FR-L-42339887'; begintime=timestamp'2013-01-16 14:55:00';
    endtime=timestamp'2013-01-16 15:05:56.288'; output;
    custnum='GB-P-87559899'; begintime=timestamp'2013-02-01 11:02:44';
    endtime=timestamp'2013-02-01 11:15:33.955'; output;
    custnum='SP-N-44333958'; begintime=timestamp'2013-03-01 10:14:33';
    endtime=timestamp'2013-03-01 10:35:27.908'; output;
    custnum='GB-R-24994990'; begintime=timestamp'2013-03-15 09:06:00';
    endtime=timestamp'2013-03-15 09:06:20.475'; output;
  end;
enddata;
run;
The following DS2 statements can be used to create table Densities in CAS. Submit the DS2 statements in the ds2.runDS2 action or in PROC DS2 with the SESSREF= option.

data densities;
  dcl char(20) Name;
  dcl double Population having format comma12.;
  dcl double SquareMiles having format comma10.;
  dcl double Density;
  method run();
    Name='Afghanistan'; Population=17070323; SquareMiles=251825; Density=67.79; output;
    Name='Albania'; Population=3407400; SquareMiles=11100; Density=306.97; output;
    Name='Algeria'; Population=28171132; SquareMiles=919595; Density=30.63; output;
    Name='Andorra'; Population=64634; SquareMiles=200; Density=323.17; output;
    Name='Angola'; Population=9901050; SquareMiles=481300; Density=20.57; output;
    Name='Antigua and Barb'; Population=65644; SquareMiles=171; Density=383.88; output;
    Name='Argentina'; Population=34248705; SquareMiles=1073518; Density=31.90; output;
    Name='Armenia'; Population=3556864; SquareMiles=11500; Density=309.29; output;
    Name='Australia'; Population=18255944; SquareMiles=2966200; Density=6.15; output;
    Name='Austria'; Population=8033746; SquareMiles=32400; Density=247.96; output;
  end;
enddata;
run;
Content

### 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';
```

### Table: Countries and Their Characteristics

<table>
<thead>
<tr>
<th>Name</th>
<th>Population</th>
<th>SquareMiles</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Afghanistan</td>
<td>17,070,323</td>
<td>251,825</td>
<td>67.790000</td>
</tr>
<tr>
<td>Albania</td>
<td>3,407,400</td>
<td>11,100</td>
<td>306.970000</td>
</tr>
<tr>
<td>Algeria</td>
<td>28,171,132</td>
<td>919,595</td>
<td>30.630000</td>
</tr>
<tr>
<td>Andorra</td>
<td>64,634</td>
<td>200</td>
<td>323.170000</td>
</tr>
<tr>
<td>Angola</td>
<td>9,901,050</td>
<td>481,300</td>
<td>20.570000</td>
</tr>
<tr>
<td>Antigua and Barb</td>
<td>65,644</td>
<td>171</td>
<td>383.880000</td>
</tr>
<tr>
<td>Argentina</td>
<td>34,248,705</td>
<td>1,073,518</td>
<td>31.900000</td>
</tr>
<tr>
<td>Armenia</td>
<td>3,556,864</td>
<td>11,500</td>
<td>309.290000</td>
</tr>
<tr>
<td>Australia</td>
<td>18,255,944</td>
<td>2,966,200</td>
<td>6.150000</td>
</tr>
<tr>
<td>Austria</td>
<td>8,033,746</td>
<td>32,400</td>
<td>247.960000</td>
</tr>
</tbody>
</table>
Hire_Date=date '2004-11-26'; output;
  EmpId=8; Dept=10; 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

<table>
<thead>
<tr>
<th>Employee Id</th>
<th>Department</th>
<th>Employee Name</th>
<th>Position</th>
<th>Hire Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>Jim Barnes</td>
<td>Manager</td>
<td>26NOV2000</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>Clifford James</td>
<td>Manager</td>
<td>26NOV2000</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>Barbara Sandman</td>
<td>Manager</td>
<td>26NOV2000</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>William Baylor</td>
<td>Manager</td>
<td>26NOV2000</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>Greg Welly</td>
<td>Developer</td>
<td>26NOV2004</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td>Penny Jackson</td>
<td>Developer</td>
<td>26NOV2004</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>Edward Murray</td>
<td>Sales Associate</td>
<td>26NOV2004</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>Ronald Thomas</td>
<td>Sales Associate</td>
<td>26NOV2004</td>
</tr>
<tr>
<td>9</td>
<td>30</td>
<td>Elsie Marks</td>
<td>Executive Assistant</td>
<td>11FEB2005</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
<td>Bruno Kramer</td>
<td>Grounds support technician</td>
<td>02NOV2005</td>
</tr>
</tbody>
</table>

Numbers1

Code

The following DS2 statements can be used to create table Numbers1 in CAS. Submit the DS2 statements in the ds2.runDS2 action or in PROC DS2 with the SESSREF= option.

data numbers1;
dcl char(5) x;
dcl double y;
dcl double z;
method run();
x='one'; y=1; z=10; output;
x='three'; y=3; z=30; output;
x='four'; y=4; z=40; output;
x='four'; y=4; z=40; output;
x='five'; y=5; z=50; output;
The following DS2 statements can be used to create table Numbers2 in CAS. Submit the DS2 statements in the ds2.runDS2 action or in PROC DS2 with the SESSREF= option.

data numbers2;
  dcl char(5) x;
  dcl double y;
  dcl double z;
  method run();
    x='four'; y=4; z=40; output;
    x='five'; y=5; z=50; output;
    x='five'; y=5; z=50; output;
    x='one'; y=1; z=10; output;
    x='two'; y=2; z=20; output;
  end;
enddata;
run;
The following DS2 statements can be used to create table Numbers3 in CAS. Submit the DS2 statements in the ds2.runDS2 action or in PROC DS2 with the SESSREF= option.

```
data numbers3;
  dcl double z;
  dcl double y;
  dcl char(5) x;
  method run();
    z=40; y=4; x='four'; output;
    z=60; y=6; x='six'; output;
    z=70; y=7; x='seven'; output;
  end;
enddata;
run;
```

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>four</td>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>five</td>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>five</td>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>one</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>two</td>
<td>2</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Z</th>
<th>Y</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>4</td>
<td>four</td>
</tr>
<tr>
<td>60</td>
<td>6</td>
<td>six</td>
</tr>
<tr>
<td>70</td>
<td>7</td>
<td>seven</td>
</tr>
</tbody>
</table>
### 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.

```ds2
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=3422; Product='Oat'; output;
    ProdId=3975; Product='Barley'; output;
end;
enddata;
run;
```

#### Content

<table>
<thead>
<tr>
<th>ProdId</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>3234</td>
<td>Rice</td>
</tr>
<tr>
<td>1424</td>
<td>Corn</td>
</tr>
<tr>
<td>3421</td>
<td>Wheat</td>
</tr>
<tr>
<td>3422</td>
<td>Oat</td>
</tr>
<tr>
<td>3975</td>
<td>Barley</td>
</tr>
</tbody>
</table>
Sales

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.

```plaintext
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>
<thead>
<tr>
<th>ProdId</th>
<th>CustId</th>
<th>Totals</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>3234</td>
<td>1</td>
<td>189400</td>
<td>United States</td>
</tr>
<tr>
<td>1424</td>
<td>3</td>
<td>555789</td>
<td>Japan</td>
</tr>
<tr>
<td>3421</td>
<td>4</td>
<td>781183</td>
<td>Japan</td>
</tr>
<tr>
<td>3421</td>
<td>2</td>
<td>2789654</td>
<td>United States</td>
</tr>
<tr>
<td>3975</td>
<td>5</td>
<td>899453</td>
<td>Argentina</td>
</tr>
</tbody>
</table>
WorldCityCoods

Code

The following DS2 statements can be used to create table WorldCityCoods 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='Shanghai'; Country='China'; Latitude=31; Longitude=121; output;
    City='Zurich'; Country='Switzerland'; Latitude=47; Longitude=8; output;
  end;
enddata;
run;
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.

```ds2
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='Shanghai'; Country='China'; AvgHigh=.; AvgLow=33; output;
```

### Content

<table>
<thead>
<tr>
<th>City</th>
<th>Country</th>
<th>AvgHigh</th>
<th>AvgLow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algiers</td>
<td>Algeria</td>
<td>90.000000</td>
<td>45.000000</td>
</tr>
<tr>
<td>Amsterdam</td>
<td>Netherlands</td>
<td>79.000000</td>
<td>33.000000</td>
</tr>
<tr>
<td>Beijing</td>
<td>China</td>
<td>86.000000</td>
<td>17.000000</td>
</tr>
<tr>
<td>Bombay</td>
<td>India</td>
<td>90.000000</td>
<td>68.000000</td>
</tr>
<tr>
<td>Calcutta</td>
<td>India</td>
<td>97.000000</td>
<td>56.000000</td>
</tr>
<tr>
<td>Caracas</td>
<td>Venezuela</td>
<td>83.000000</td>
<td>57.000000</td>
</tr>
<tr>
<td>Geneva</td>
<td>Switzerland</td>
<td>76.000000</td>
<td>28.000000</td>
</tr>
<tr>
<td>Hong Kong</td>
<td>China</td>
<td>89.000000</td>
<td>51.000000</td>
</tr>
<tr>
<td>Lagos</td>
<td>Nigeria</td>
<td>90.000000</td>
<td>75.000000</td>
</tr>
<tr>
<td>Madrid</td>
<td>Spain</td>
<td>89.000000</td>
<td>36.000000</td>
</tr>
<tr>
<td>Shanghai</td>
<td>China</td>
<td>.</td>
<td>33.000000</td>
</tr>
<tr>
<td>Zurch</td>
<td>Switzerland</td>
<td>78.000000</td>
<td>25.000000</td>
</tr>
</tbody>
</table>
COPYRIGHT AND PERMISSION NOTICE

Copyright (c) 1995-2005 International Business Machines Corporation and others
All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this
software and associated documentation files (the "Software"), to deal in the
Software without restriction, including without limitation the rights to use, copy,
modify, merge, publish, distribute, and/or sell copies of the Software, and to permit
persons to whom the Software is furnished to do so, provided that the above
copyright notice(s) and this permission notice appear in all copies of the Software
and that both the above copyright notice(s) and this permission notice appear in
supporting documentation.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE
COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE
FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL
DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used
in advertising or otherwise to promote the sale, use or other dealings in this
Software without prior written authorization of the copyright holder.

All trademarks and registered trademarks mentioned herein are the property of their
respective owners.