



THE
POWER
TO KNOW.

SAS[®] 9.4 FedSQL Language Reference

Third Edition

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2014. *SAS® 9.4 FedSQL Language Reference, Third Edition*. Cary, NC: SAS Institute Inc.

SAS® 9.4 FedSQL Language Reference, Third Edition

Copyright © 2014, SAS Institute Inc., Cary, NC, USA

All rights reserved. Produced in the United States of America.

For a hard-copy book: No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

For a web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

U.S. Government License Rights; Restricted Rights: The Software and its documentation is commercial computer software developed at private expense and is provided with RESTRICTED RIGHTS to the United States Government. Use, duplication or disclosure of the Software by the United States Government is subject to the license terms of this Agreement pursuant to, as applicable, FAR 12.212, DFAR 227.7202-1(a), DFAR 227.7202-3(a) and DFAR 227.7202-4 and, to the extent required under U.S. federal law, the minimum restricted rights as set out in FAR 52.227-19 (DEC 2007). If FAR 52.227-19 is applicable, this provision serves as notice under clause (c) thereof and no other notice is required to be affixed to the Software or documentation. The Government's rights in Software and documentation shall be only those set forth in this Agreement.

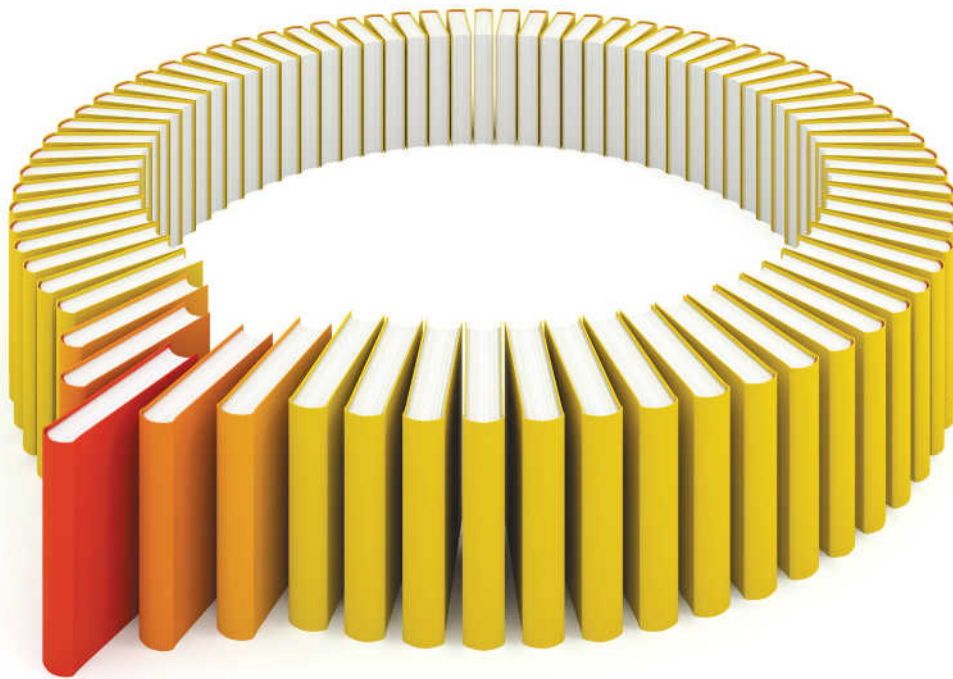
SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513-2414.

August 2014

SAS provides a complete selection of books and electronic products to help customers use SAS® software to its fullest potential. For more information about our offerings, visit support.sas.com/bookstore or call 1-800-727-3228.

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.



Gain Greater Insight into Your SAS[®] Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

 support.sas.com/bookstore
for additional books and resources.


THE POWER TO KNOW.

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are trademarks of their respective companies. © 2013 SAS Institute Inc. All rights reserved. S107969US.0613

Contents

<i>What's New in SAS 9.4 FedSQL Language Reference</i>	<i>ix</i>
<i>Recommended Reading</i>	<i>xi</i>

PART 1 Introduction 1

Chapter 1 • Using This Language Reference	3
Purpose	3
Intended Audience	3
Chapter 2 • Conventions	5
Typographical Conventions	5
Syntax Conventions	5

PART 2 FedSQL Language Reference 7

Chapter 3 • FedSQL Language Concepts	9
Introduction to the FedSQL Language	10
Running FedSQL Programs	10
Data Source Support	11
Benefits of FedSQL	11
Federated Queries	12
Data Source Connection	12
Data Types	13
Identifiers	17
How FedSQL Processes Nulls and SAS Missing Values	18
Type Conversions	23
NLS Transcoding Failures	26
Join Operations	27
FedSQL Expressions	42
Dates and Times in FedSQL	51
DICTIONARY Tables	58
FedSQL Pass-Through Facility	59
FedSQL Implicit Pass-Through Facility	60
Transactions in FedSQL	61
FedSQL Reserved Words	61
Chapter 4 • FedSQL Formats	67
Overview of Formats	69
General Format Syntax	69
Using Formats in FedSQL	70
Validation of FedSQL Formats	71
FedSQL Format Examples	71
Using a User-Defined Format	72
SAS Output Delivery System and FedSQL	73
Format Categories	73

NLS Formats Supported by FedSQL	73
Formats Supported with the PUT Function, by Category	76
Dictionary	81
Chapter 5 • FedSQL Functions	177
Overview of FedSQL Functions	179
General Function Syntax	180
Using FedSQL Functions	181
Aggregate Functions	182
Function Categories	183
FEDSQL Functions by Category	184
Dictionary	190
Chapter 6 • FedSQL Expressions and Predicates	321
Overview of Expressions and Predicates	321
Dictionary	322
Chapter 7 • FedSQL Informats	345
Definition of an Informat	345
General Informat Syntax	345
How Informats Are Used in FedSQL	346
How to Specify Informats in FedSQL	347
Validation of FedSQL Informats	347
FedSQL Informat Example	347
Chapter 8 • FedSQL Statements	349
Overview of Statements	349
FedSQL Statements by Category	350
Dictionary	351
Chapter 9 • FedSQL Statement Table Options	415
Overview of Statement Table Options	417
How Table Options Interact with Other Types of Options	417
FedSQL Statement Table Option Syntax	418
Understanding BULKLOAD Table Options	418
FedSQL Statement Table Options by Data Source	418
Dictionary	426
 PART 3 Appendixes 501	
Appendix 1 • FedSQL and the ANSI Standard	503
Compliance	503
FedSQL Enhancements	503
FedSQL Limitations	505
Appendix 2 • Data Type Reference	507
Data Types for SAS Data Sets	507
Data Types for SPD Engine Data Sets	509
Data Types for Aster	511
Data Types for DB2 under UNIX and PC Hosts	512
Data Types for Greenplum	513
Data Types for HDMD	514
Data Types for Hive	516
Data Types for MDS	517

Data Types for MySQL	519
Data Types for Netezza	521
Data Types for ODBC	522
Data Types for Oracle	523
Data Types for PostgreSQL	525
Data Types for SAP	527
Data Types for SAP HANA	529
Data Types for SASHDAT	530
Data Types for Sybase IQ	531
Data Types for Teradata	533
Appendix 3 • Using FedSQL and DS2	535
Appendix 4 • Tables Used in Examples	537
AfewWords	537
Customers	538
CustonLine	538
Densities	539
Depts	539
Employees	540
GrainProducts	540
Integers	541
Products	541
Sales	541
WorldCityCoords	542
WorldTemps	542
Appendix 5 • DICTIONARY Table Descriptions	545
DICTIONARY.CATALOGS	545
DICTIONARY.COLUMNS	545
DICTIONARY.COLUMN STATISTICS	549
DICTIONARY.STATISTICS	550
DICTIONARY.TABLES	555
Appendix 6 • Usage Notes	557
Appendix 7 • ICU License	559
ICU License - ICU 1.8.1 and later	559
Third-Party Software Licenses	560
Index	567

What's New in SAS 9.4 FedSQL Language Reference

Introduction to the FedSQL Language

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

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 table.

Documentation Enhancements

If you previously used the FedSQL language with DataFlux Federation Server, *SAS FedSQL Language Reference* replaces *DataFlux Federation Server FedSQL Reference Guide*. The FedSQL language is available now for Base SAS users as well as for users of SAS Federation Server.

First Maintenance Release of SAS 9.4

In the first maintenance release of SAS 9.4, the following functionality was added to the FedSQL language:

- Support for Memory Data Store (MDS), SAP HANA, and SASHDAT data sources.
- Rename table and rename column functionality in the ALTER TABLE statement. See [“ALTER TABLE Statement” on page 351](#).

The documentation was enhanced to include the following:

- examples of join operations. See [“Join Operations” on page 27](#).
- examples of EXCEPT, INTERSECT, and UNION operations. See [“Examples of Query Expressions” on page 42](#).

- examples of subqueries. See [“Overview of Subqueries” on page 44](#), [“Examples of Correlated Subqueries” on page 45](#), [“Examples of Scalar Subqueries” on page 46](#), and [“Examples of Non-Correlated Queries” on page 48](#).
- an explanation of how the PUT function can be used. See [“Using Formats in FedSQL” on page 70](#) and [“Using a User-Defined Format” on page 72](#).
- an explanation of how FedSQL handles transactions. See [“Transactions in FedSQL” on page 61](#).
- reference information for additional functions. See [Chapter 5, “FedSQL Functions,” on page 177](#).

Second Maintenance Release of SAS 9.4

In the second maintenance release of SAS 9.4, the following functionality was added to the FedSQL language:

- Support for Hive, HDMD, and PostgreSQL data sources. See [“Data Types for Hive” on page 516](#), [“Data Types for HDMD” on page 514](#), and [“Data Types for PostgreSQL” on page 525](#) to see what data types are supported for each data source. Not all FedSQL statements are supported for each data source. See the documentation for FedSQL statements to determine statement support.
- [“CAST Function” on page 210](#)
- [“DBCREATE_INDEX_OPTS=” on page 450](#) for ODBC
- [“SQUEEZE= Table Option” on page 476](#) for SASHDAT

Recommended Reading

- *SAS Federation Server: Administrator's Guide*
- *SAS DS2 Language Reference*
- *SAS LIBNAME Engine for SAS Federation Server: User's Guide*
- *SAS System Options: Reference*
- *SAS Formats and Informats: Reference*
- *SAS National Language Support (NLS): Reference Guide*
- *Encryption in SAS*

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

SAS Books
SAS Campus Drive
Cary, NC 27513-2414
Phone: 1-800-727-3228
Fax: 1-919-677-8166
E-mail: sasbook@sas.com
Web address: support.sas.com/bookstore

Part 1

Introduction

<i>Chapter 1</i>	
Using This Language Reference	3
<i>Chapter 2</i>	
Conventions	5

Chapter 1

Using This Language Reference

Purpose	3
Intended Audience	3

Purpose

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

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

Intended Audience

This document is intended for the following users:

- Application developers who write client applications need an understanding of the FedSQL language. They write applications that create tables, bulk load tables, manipulate tables, and query data.
- Database administrators who design and implement the client/server environment. They administer the data by designing the databases and setting up the data source metadata. That is, database administrators build the data model.
- SAS programmers who write and submit Base SAS code such as SAS procedures and the SAS DATA step language and now want to take advantage of the FedSQL language.

Chapter 2

Conventions

Typographical Conventions	5
Syntax Conventions	5

Typographical Conventions

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

UPPERCASE BOLD

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

UPPERCASE ROMAN

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

italic

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

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

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

monospace

identifies examples of SAS code.

Syntax Conventions

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

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

syntax. In the FedSQL syntax, square brackets ([]) are used to denote optional syntax and angle brackets are used to denote non-terminal components.

The following symbols are used in the FedSQL syntax.

::=

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

<>

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

[]

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

{ }

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

|

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

...

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

\

A backward slash indicates that next character is a literal.

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

```

1 CREATE TABLE 2 {table | _NULL_}
3 [{\{OPTIONS SAS-table-option=value [... SAS-table-option=value]\}}]
4 <column-definition> 5 [, ...<column-definition> | <table-constraint>])
6 | 7 AS query-expression
;
<column-definition>::=
    column data-type [<column-constraint>] [DEFAULT value]
    [HAVING [FORMAT format] [INFORMAT informat] [LABEL='label']]

```

- 1** **CREATE TABLE** is in uppercase bold because it is the name of the statement.
- 2** *table* is italics because it is an argument that you can supply.
- 3** The braces are preceded by a backward slash indicating that the braces are literal.
- 4** <column-definition> is in angle brackets because it is a non-terminal argument that is further resolved into lower level syntax grammar.
- 5** The square brackets and ellipsis around the second instance of <column-definition> indicates that it is optional and you can repeat this argument any number of times separated by commas.
- 6** You can supply a <column-definition> or AS *query-expression* but not both.
- 7** AS is in uppercase roman because it is a literal argument.

Part 2

FedSQL Language Reference

<i>Chapter 3</i>	
FedSQL Language Concepts	9
<i>Chapter 4</i>	
FedSQL Formats	67
<i>Chapter 5</i>	
FedSQL Functions	177
<i>Chapter 6</i>	
FedSQL Expressions and Predicates	321
<i>Chapter 7</i>	
FedSQL Informats	345
<i>Chapter 8</i>	
FedSQL Statements	349
<i>Chapter 9</i>	
FedSQL Statement Table Options	415

Chapter 3

FedSQL Language Concepts

Introduction to the FedSQL Language	10
Running FedSQL Programs	10
Data Source Support	11
Benefits of FedSQL	11
Federated Queries	12
Data Source Connection	12
Data Types	13
Identifiers	17
Overview of Identifiers	17
Regular Identifiers	17
Delimited Identifiers	18
Support for Non-Latin Characters	18
How FedSQL Processes Nulls and SAS Missing Values	18
FedSQL Modes for Nonexistent Data	18
Differences between Processing Null Values and SAS Missing Values	19
Reading and Writing Nonexistent Data in ANSI Mode	20
Reading and Writing Nonexistent Data in SAS Mode	21
Testing and Modifying Nulls and SAS Missing Values	22
Type Conversions	23
Type Conversion Definitions	23
Overview of Type Conversions	23
Type Conversion for Unary Expressions	24
Type Conversion for Logical Expressions	24
Type Conversion for Arithmetic Expressions	25
Type Conversion for Relational Expressions	25
Type Conversion for Concatenation Expressions	26
NLS Transcoding Failures	26
Join Operations	27
Overview of Join Operations	27
Understanding the Join Operations	28
Inner and Outer Join Types	37
Joining Heterogeneous Data	40
FedSQL Expressions	42
Query Expressions and Subqueries	42
FedSQL Value Expressions	50

Dates and Times in FedSQL	51
Overview of Dates and Times in FedSQL	51
FedSQL Date, Time, and Datetime Constants	52
Using Dates and Times in SAS Data Sets and SPD Engine Data Sets	52
Date, Time, and Datetime Functions	54
Date, Time, and Datetime Formats for SAS Data Sets and SPD Engine Data Sets	54
DICTIONARY Tables	58
FedSQL Pass-Through Facility	59
Overview	59
CONNECTION TO Component of the FROM Clause	59
EXECUTE Statement	59
FedSQL Implicit Pass-Through Facility	60
Overview	60
How to Use FedSQL Implicit Pass-Through	60
Single Source FedSQL Implicit Pass-Through	60
Multiple Source FedSQL Implicit Pass-Through	60
Transactions in FedSQL	61
FedSQL Reserved Words	61

Introduction to the FedSQL Language

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

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

For more information about SQL:1999 core-compliant syntax and SAS extensions, see [“FedSQL and the ANSI Standard” on page 503](#).

Running FedSQL Programs

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

- From the Base SAS language interface by using the FEDSQL procedure. See *Base SAS Procedures Guide*.
- From a JDBC, ODBC, or OLE DB client by using SAS Federation Server. See *SAS Federation Server: Administrator's Guide*.
- From a Base SAS language interface by using SAS Federation Server LIBNAME engine and the PROC SQL pass-through facility. You can specify FedSQL statements in the EXECUTE statement in the same way that you execute DBMS-

specific SQL statements. See *SAS LIBNAME Engine for SAS Federation Server: User's Guide*.

- From a DS2 program. See [Appendix 3, “Using FedSQL and DS2,” on page 535](#).
- From the SPD Server by using the SQL pass-through facility or through an ODBC or JDBC client. See *SAS Scalable Performance Data Server: User's Guide*

Data Source Support

FedSQL can access the following data sources:

- Aster
- DB2 for UNIX and PC operating environments
- Greenplum
- Hadoop (Hive and HDMD)
- Memory Data Store (MDS)
- MySQL
- Netezza
- ODBC databases (such as Microsoft SQL Server)
- Oracle
- PostgreSQL
- SAP (Read-only)
- SAP HANA
- SASHDAT files
- Sybase IQ
- SAS data sets
- SAS Scalable Performance Data Engine (SPD Engine) data sets
- Teradata

Note: The procedures and SAS Federation Server support different data sources. See *Base SAS Procedures Guide* and *SAS Federation Server: Administrator's Guide* for information about the data sources that each one supports.

Benefits of FedSQL

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

- FedSQL conforms to the ANSI SQL:1999 core standard. This conformance allows it to process queries in its own language and the native languages of other data sources that conform to the standard.
- FedSQL supports many more data types than previous SAS SQL implementations. Traditional data source access through SAS/ACCESS translates target data source

data types to and from two legacy SAS data types, which are SAS numeric and SAS character. When FedSQL connects to a data source, the language matches or translates the target data source's definition to these data types, as appropriate, which allows greater precision. Supported data types are described in [“Data Types” on page 13](#).

- FedSQL handles federated queries. With the traditional DATA step or the SQL procedure, a SAS/ACCESS LIBNAME engine can access only the data of its intended data source.
- The FedSQL language can create data in any of the supported data sources, even if the target data source is not represented in a query. This enables you to store data in the data source that most closely meets the needs of your application.

Federated Queries

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

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

Data Source Connection

In order to connect to a data source, the FedSQL language requires that a connection string be submitted that defines how to connect to a data source.

The FEDSQL procedure generates a connection string by using the attributes of currently assigned librefs. You first submit a LIBNAME statement for the data source that you want to access (for example, a Base SAS LIBNAME statement or a Hadoop LIBNAME statement) and then submit PROC FEDSQL. For more information, see Chapter 24, “FEDSQL Procedure” in *Base SAS Procedures Guide*.

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

SAS Scalable Performance Data Server generates a connection string to the FedSQL language when you use the SQL pass-through facility and set the SQL dialect using the EXECUTE statement:

```
Execute(reset dialect=fedsql)
```

See *SAS Scalable Performance Data Server: User's Guide*.

SAS Federation Server encapsulates the information that is needed to connect to a data source in a Data Source Name (DSN) definition. A DSN is metadata that contains connection details. DSN definitions are created by a SAS Federation Server administrator. In order to use a DSN, you must first connect to SAS Federation Server.

For more information about connecting to SAS Federation Server and submitting DSNs, see *SAS Federation Server: Administrator's Guide*.

Data Types

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

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

Table 3.1 FedSQL Data Types

Data Type	Description
BIGINT	stores a large signed, exact whole number, with a precision of 19 digits. The range of integers is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Integer data types do not store decimal values; fractional portions are discarded.
BINARY(<i>n</i>)	stores fixed-length binary data, where <i>n</i> is the maximum number of bytes to store. The maximum number of bytes is required to store each value regardless of the actual size of the value.
CHAR(<i>n</i>)	stores a fixed-length character string, where <i>n</i> is the maximum number of characters to store. The maximum number of characters is required to store each value regardless of the actual size of the value. If char (10) is specified and the character string is only five characters long, the value is right-padded with spaces.
DATE	<p>stores a calendar date. A date literal is specified in the format <i>yyyy-mm-dd</i>: a four-digit year (0001 to 9999), a two-digit month (01 to 12), and a two-digit day (01 to 31). For example, the date September 24, 1975 is specified as 1975-09-24.</p> <p>FedSQL complies with ANSI SQL:1999 standards regarding dates. However, not all data sources support the full range of dates. For example, dates between 0001-01-01 and 1582-12-31 are not valid dates for a SAS data set and an SPD Engine data set.</p>

Data Type	Description
DOUBLE	stores a signed, approximate, double-precision, floating-point number. Allows numbers of large magnitude and permits computations that require many digits of precision to the right of the decimal point.
DECIMAL NUMERIC(<i>p,s</i>)	<p>stores a signed, exact, fixed-point decimal number, with user-specified precision and scale. The precision and scale determine the position of the decimal point. The precision is the maximum number of digits that can be stored to the left and right of the decimal point, with a range of 1 to 52. The scale is the maximum number of digits that can be stored following the decimal point. The scale must be less than or equal to the precision. For example, decimal(9,2) stores decimal numbers up to nine digits, with a two-digit, fixed-point fractional portion, such as 1234567.89.</p> <p><i>Note:</i> The DECIMAL data type is supported for defining a column, inserting data into the column, and fetch operations. Other operations, such as using a DECIMAL column in an expression, result in the DECIMAL data type being converted to a DOUBLE.</p>
FLOAT(<i>p</i>)	stores a signed, approximate, single-precision or double-precision, floating-point number. The user-specified precision determines whether the data type stores a single-precision or double-precision number. If the specified precision is equal to or greater than 25, the value is stored as a double-precision number, which is a DOUBLE. If the specified precision is less than 25, the value is stored as a single-precision number, which is a REAL. For example, float(10) specifies to store up to 10 digits, which results in a REAL data type.
INTEGER	<p>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.</p> <p><i>Note:</i> Integer division by zero does not produce the same result on all operating systems. It is recommended that you avoid integer division by zero.</p>

Data Type	Description
NCHAR(<i>n</i>)	stores a fixed-length character string such as CHAR but uses a Unicode national character set, where <i>n</i> is the maximum number of multibyte characters to store. Depending on the platform, Unicode characters use either two or four bytes per character and support all international characters.
NVARCHAR(<i>n</i>)	stores a varying-length character string such as VARCHAR but uses a Unicode national character set, where <i>n</i> is the maximum number of multibyte characters to store. Depending on the platform, Unicode characters use either two or four bytes per character and can support all international characters.
REAL	stores a signed, approximate, single-precision, floating-point number.
SMALLINT	stores a small signed, exact whole number, with a precision of five digits. The range of integers is -32,768 to 32,767. Integer data types do not store decimal values; fractional portions are discarded.
TIME(<i>p</i>)	stores a time value. A time literal is specified in the format <i>hh:mm:ss[.nnnnnnnnn]</i> ; a two-digit hour 00 to 23, a two-digit minute 00 to 59, and a two-digit second 00 to 61 (supports leap seconds), with an optional fraction value. For example, the time 6:30 a.m. is specified as 06:30:00 . When supported by a data source, the <i>p</i> parameter specifies the seconds precision. The seconds precision is an optional fraction value that is up to nine digits long.
TIMESTAMP(<i>p</i>)	stores both date and time values. A timestamp literal is specified in the format <i>yyyy-mm-dd:hh:mm:ss[.nnnnnnnnn]</i> : a four-digit year 0001 to 9999, a two-digit month 01 to 12, a two-digit day 01 to 31, a two-digit hour 00 to 23, a two-digit minute 00 to 59, and a two-digit second 00 to 61 (supports leap seconds), with an optional fraction value. For example, the date and time September 24, 1975 6:30 a.m. is specified as 1975-09-24:06:30:00 . When supported by a data source, the <i>p</i> parameter specifies the seconds precision. The seconds precision is an optional fraction value that is up to nine digits long.

Data Type	Description
TINYINT	stores a very small signed, exact whole number, with a precision of three digits. The range of integers is -128 to 127. Integer data types do not store decimal values; fractional portions are discarded.
VARBINARY(<i>n</i>)	stores varying-length binary data, where <i>n</i> is the maximum number of bytes to store. The maximum number of bytes is not required to store each value. If varbinary(10) is specified and the binary string uses only five bytes, only five bytes are stored in the column.
VARCHAR(<i>n</i>)	stores a varying-length character string, where <i>n</i> is the maximum number of characters to store. The maximum number of characters is not required to store each value. If varchar(10) is specified and the character string is only five characters long, only five characters are stored in the column.

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

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

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

For details about data source implementation for each data type, see [“Data Type Reference” on page 507](#).

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

Identifiers

Overview of Identifiers

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

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

Regular identifiers are the type of identifiers that you see in most programming languages. They are not case-sensitive; the identifier “Name” is the same as “NAME” and “name”. Only certain characters are allowed in regular identifiers.

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

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

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

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

Regular Identifiers

When you name regular identifiers, use these rules:

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

The following are valid regular identifiers:

```
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`.

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

Delimited Identifiers

When you name delimited identifiers, follow these rules:

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

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

The following 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 61](#).

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

Support for Non-Latin Characters

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

How FedSQL Processes Nulls and SAS Missing Values

FedSQL Modes for Nonexistent Data

Nonexistent data is represented by a SAS missing value in SAS data sets and SPD Engine data sets. For all other data sources, nonexistent data is represented by an ANSI SQL null value. The SAS missing value indicators, . , ._, .A - .Z, and ' _' are known

values that indicate nonexistent data. Table data with an ANSI null has no real data value; it is metadata that indicates an unknown value.

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

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

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

In most instances, no mode change is necessary to process nonexistent data. The following are instances of when you might want to change the mode:

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

CAUTION:

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

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

Differences between Processing Null Values and SAS Missing Values

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

- when filtering data (for example, in a WHERE clause, a HAVING clause, or an outer join ON clause). SAS mode interprets null values as SAS missing values, which are known values, whereas ANSI mode interprets null values as unknown values.
- when submitting outer joins in ANSI mode, internal processing might generate nulls for intermediate result sets. FedSQL might generate SAS missing values in SAS mode for intermediate result sets. Therefore, for intermediate result sets, nulls are interpreted as unknown values in ANSI mode and in SAS mode, missing values are interpreted as known values.
- when comparing a blank character. SAS mode interprets the blank character as a missing value. In ANSI mode, a blank character is a blank character; it has no special meaning.

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

Table 3.2 Attribute and Behavior Differences between Null Values and SAS Missing Values

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

For information about the results of logical operations on null values, see “<search-condition>” on page 410.

Reading and Writing Nonexistent Data in ANSI Mode

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

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

Table 3.3 Reading Nonexistent Data Values in ANSI Mode

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

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

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

Table 3.4 Writing Nonexistent Data Values in ANSI Mode

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

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

Reading and Writing Nonexistent Data in SAS Mode

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

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

Table 3.5 Reading Nonexistent Data Values in SAS Mode

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

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

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

Table 3.6 Writing Nonexistent Data Values in SAS Mode

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

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

Testing and Modifying Nulls and SAS Missing Values

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

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

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

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

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

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

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

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

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

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

For more information, see [“IFNULL Function” on page 246](#) and [“NULLIF Expression” on page 338](#).

Type Conversions

Type Conversion Definitions

- binary data type
refers to the VARBINARY and BINARY data type.
- character data type
refers to the CHAR, VARCHAR, NCHAR, and NVARCHAR data types.
- coercible data type
a data type that can be converted to multiple data types, not just a character data type.
- date/time data type
refers to the DATE, TIME, and TIMESTAMP data types.
- non-coercible data type
a data type that can be converted only to a character data type.
- numeric data type
refers to the DECIMAL, NUMERIC, DOUBLE (or FLOAT), REAL, BIGINT, INT, SMALLINT, and TINYINT data types.
- standard character conversion
if an expression is not one of the character data types, it is converted to a CHAR data type.
- standard numeric conversion
if an expression has a coercible, non-numeric data type, it is converted to a DOUBLE data type.

Overview of Type Conversions

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

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

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

When FedSQL evaluates an expression, if the data types of the operands match exactly, no type conversion or promotion is necessary and the expression is resolved. Otherwise,

each operand must go through a standard numeric conversion or a standard character conversion, depending on the operator.

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

Type Conversion for Unary Expressions

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

Table 3.7 Data Type Conversion for Unary Expressions

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

Type Conversion for Logical Expressions

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

Table 3.8 Data Type Conversion for Logical Expressions

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

Type Conversion for Arithmetic Expressions

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

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

Table 3.9 Type Conversion for Addition, Subtraction, Multiplication, and Division Expressions

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

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

Table 3.10 Data Type Conversion for the Min, Max, and Power Operator Expressions

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

Type Conversion for Relational Expressions

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

Table 3.11 Data Type Conversion for Relational Expressions except IN Expressions

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

Table 3.12 Data Type Conversion for IN Expressions

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

Type Conversion for Concatenation Expressions

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

Table 3.13 Data Type Conversion for Concatenation Expressions

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

NLS Transcoding Failures

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

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

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

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

Join Operations

Overview of Join Operations

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

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

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

Output 3.1 Join Result Sets of Tables Products and Sales

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

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

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

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

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

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

Note: The join operation examples in this section use the tables Customers, Products, and Sales. To view the tables, see [“Tables Used in Examples” on page 537](#).

Understanding the Join Operations

Simple Join

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

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

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

Output 3.3 Simple Join of Two Tables

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

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

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

Output 3.4 Simple Join of Three Tables

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

Equijoin

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

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

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

Output 3.5 *Equijoin of All Columns*

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

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

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

Output 3.6 *Equijoin with Specified Columns*

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

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

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

Output 3.7 *Equijoin of Three Tables*

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

Cross Join

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

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

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

Output 3.8 Cross Join of Two Tables

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

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

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

Output 3.9 Cross Join with a WHERE Clause

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

Qualified Join

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

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

A qualified join can be an inner join or an outer join, which is requested with the syntax INNER or OUTER. If the join type specification is omitted, then an inner join is implied. See [“Inner and Outer Join Types”](#) on page 37.

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

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

Output 3.10 Qualified Join with an ON Clause

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

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

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

Output 3.11 Qualified Join with a USING Clause

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

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

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

Output 3.12 Qualified Join with an ON Clause and a WHERE Clause

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

Natural Join

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

A natural join can be an inner join or an outer join, which is requested with the syntax `INNER` or `OUTER`. If the join type specification is omitted, then an inner join is implied. See “[Inner and Outer Join Types](#)” on page 37.

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

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

Output 3.13 Natural Join of All Columns

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

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

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

Output 3.14 Natural Join with a WHERE Clause

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

Inner and Outer Join Types

Overview of Inner and Outer Join Types

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

Inner Joins

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

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

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

Outer Joins

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

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

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

```
select customers.city, sales.totals
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Joining Heterogeneous Data

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

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

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

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

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

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

Output 3.15 FedSQL Heterogeneous Join of a SAS Data Set and an Oracle Table

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

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

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

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

Output 3.16 FedSQL Heterogeneous Join of Two Oracle Tables and a SAS Data Set

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

FedSQL Expressions

Query Expressions and Subqueries

Overview of Query Expressions

A *query expression* or *query* is one or more SELECT statements that produce a result set. Multiple SELECT statements can be combined by set operators.

Set operators (UNION, EXCEPT, and INTERSECT) combine columns from two queries based on their position in the referenced tables without regard to individual column names. Columns in the same relative position in the two queries must have the same data types. The column names of the tables in the first query become the column names of the result set.

A query expression with set operators is evaluated as follows.

- Each SELECT statement is evaluated to produce a virtual, intermediate result set.
- Each intermediate result set then becomes an operand that is linked with a set operator to form an expression (for example, A UNION B).
- If the query expression involves more than two SELECT statements, then the result from the first two becomes an operand for the next set operator and operand, such as (A UNION B) EXCEPT C, ((A UNION B) EXCEPT C) INTERSECT D, and so on.
- Evaluating a query expression produces a single output result set.

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.

Examples of Query Expressions

To understand how query expressions work, consider the following examples. The examples operate on two tables, named Numbers1 and Numbers2.

The following displays show the content of the tables.

Display 3.1 Content of numbers1 Table

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

Display 3.2 Content of numbers2 Table

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

The following example code specifies the EXCEPT operator:

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

The EXCEPT operator returns values that exist in one table but not the other table in the comparison.

Output 3.17 Output from EXCEPT Operation

X	Y	Z
three	3	30

The following example code specifies the INTERSECT operator:

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

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

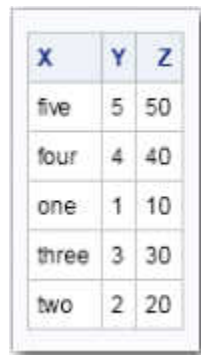
Output 3.18 Output from the INTERSECT Operation


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

The following example code specifies the UNION operator:

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

The UNION operator merges the unique values in the specified tables to display as one table.

Output 3.19 Output from the UNION Operation


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

For more information about using set operators, see the “SELECT Statement” on page 386.

Overview of Subqueries

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

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

Subqueries can appear in various places within a query:

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

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

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

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

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

Examples of Correlated Subqueries

The following is an example of a correlated subquery that specifies the EXISTS predicate. It uses data from the example Employees and Depts tables. For a description of these tables, see “Employees” on page 540 and “Depts” on page 539.

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

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

Output 3.20 Output of Correlated Query with EXISTS Predicate

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

Here is a more efficient way to write this query:

```
select *
from employees
where dept in (select deptno from depts)
```

```
where pos <> 'Manager');
```

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

Examples of Scalar Subqueries

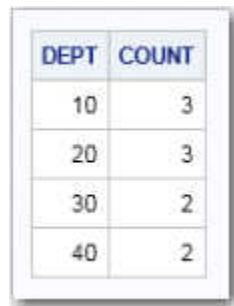
A scalar subquery returns one value for one column. As such, it is useful for aggregate queries.

Here is an example of a scalar subquery in the HAVING clause. The example uses the Employees and Depts tables that were used in [“Examples of Correlated Subqueries” on page 45](#).

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

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

Output 3.21 Number of Employees by Department



DEPT	COUNT
10	3
20	3
30	2
40	2

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

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

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

Output 3.22 Number of Employees by Manager

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

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

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

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

Output 3.23 Departments with More Than One Employee

DEPTNO	DEPTNAME	MANAGER
10	Sales	Jim Barnes
20	Research	Clifford James

The following is an example of a scalar subquery in the INSERT statement. This example uses data from the Densities example table, which has a column named Population. A new table, Summary, is created and populated with aggregated values from Densities table's Population column. For more information about the Densities table, see [“Densities” on page 539](#).

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

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

```

        (select avg(population) from densities)
    );
select * from summary;

```

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

Output 3.24 Content of the Summary Table

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

Examples of Non-Correlated Queries

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

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

Output 3.25 Table All1

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

Output 3.26 Table One

x
1

Output 3.27 Table Two

x	c
2	two

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

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

Output 3.28 Output of the SELECT Subquery



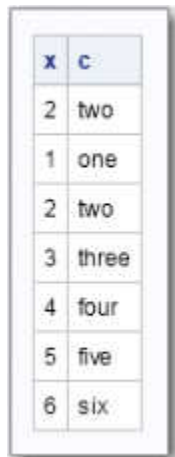
x	c
1	one

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

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

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

Output 3.29 Results of the INSERT Subquery



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

This example code specifies a subquery in the DELETE statement:

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

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

Output 3.30 Results of the DELETE Subquery


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

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

```
update all1 set x = (select x from two where c = 'one');
select * from all1;
```

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

Output 3.31 Results of the UPDATE Subquery


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

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

FedSQL Value Expressions

Numeric Value Expressions

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

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

Here are examples of numeric value expressions.

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

Row Value Expressions

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

A row value constructor can contain the following values.

- *value-expression*
- NULL
- DEFAULT
- *row-subquery*
- (*row-value-constructor*, *row-value-constructor*, ...)
- ROW (*row-value-constructor*, *row-value-constructor*, ...)

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

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

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

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

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

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

Dates and Times in FedSQL

Overview of Dates and Times in FedSQL

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

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

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

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

FedSQL Date, Time, and Datetime Constants

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

DATE 'yyyy-mm-dd'

TIME 'hh: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
- *nn* is a two-digit minute, 00–59
- *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 10.

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

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

The following are examples of FedSQL date, time, and timestamp constants:

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

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

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

Using Dates and Times in SAS Data Sets and SPD Engine Data Sets

When you create a table using the BASE driver or the SPD drivers, you specify columns for date, time, and datetime values by using the DATE, TIME, and TIMESTAMP data

types. FedSQL converts values with these data types to a data type of DOUBLE having these SAS formats:

Table 3.14 SAS Formats Assigned to Date and Time Values in SAS Data Sets and SPD Engine Data Sets

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

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

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

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

Here is the output:

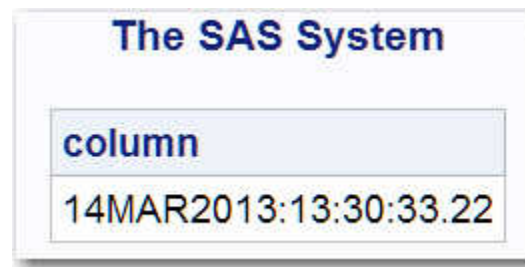
Output 3.32 Results for Date, Time, and Timestamp

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

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

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

Here is the output:

Output 3.33 Results of Using the PUT Statement

Date, Time, and Datetime Functions

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

Table 3.15 Date, Time, and Datetime Functions

Function	Description
CURRENT_DATE	returns the current date for your time zone.
CURRENT_TIME	returns the current time for your timezone.
CURRENT_TIME_GMT()	returns the current GMT time.
CURRENT_TIMESTAMP	returns the current date and time for your timezone.
CURRENT_TIMESTAMP_GMT()	returns the current GMT date and time.
DAY	returns the numeric day of the month from a date or datetime value.
HOUR	returns the hour from a time or datetime value.
MINUTE	returns the minute from a time or datetime value.
MONTH	returns the numeric month from a date or datetime value.
SECOND	returns the second from a time or datetime value.
YEAR	returns the year from a date or datetime value.

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

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

Table 3.16 Date, Time, and Datetime Functions Arranged by Type

Type	Language Element	Input	Formatted Result
Date formats	DATE.	date'2013-03-17'	17MAR13
	DATE9.	date'2013-03-17'	17MAR2013
	DATEAMP21.	timestamp'2013-03-17 14:22:21'	17MAR13:02:22:21 PM
	DATEAMP23.	timestamp'2013-03-17 14:22:21'	17MAR2013:02:22:21 PM
	DAY.	date'2013-03-17'	17
	DDMMYY.	date'2013-03-17'	17/03/13
	DDMMYYB.	date'2013-03-17'	17 03 13
	DDMMYYC.	date'2013-03-17'	17:03:13
	DDMMYYD10.	date'2013-03-17'	17-03-2013
	DDMMYYN.	date'2013-03-17'	17032013
	DDMMYYYP.	date'2013-03-17'	17.03.13
	DDMMYYYS.	date'2013-03-17'	17/03/13
	DOWNAME.	date'2013-03-17'	Monday
	JULIAN.	date'2013-03-17'	13077
	MMDDYY.	date'2013-03-17'	03/17/13
	MMDDYY10.	date'2013-03-17'	03/17/2013
	MMDDYYB.	date'2013-03-17'	03 17 13
	MMDDYC.	date'2013-03-17'	03:17:13
	MMDDYD.	date'2013-03-17'	03-17-13
	MMDDYN.	date'2013-03-17'	03172013
	MMDDYYP.	date'2013-03-17'	03.17.13
	MMDDYYYS.	date'2013-03-17'	03/17/13
	MMYY.	date'2013-03-17'	03M2013

Type	Language Element	Input	Formatted Result
	MMYYC.	date'2013-03-17'	03:2013
	MMYYD.	date'2013-03-17'	03-2013
	MMYYN.	date'2013-03-17'	032013
	MMYYP.	date'2013-03-17'	03.2013
	MMYYs.	date'2013-03-17'	03/2013
	MONNAME.	date'2013-03-17'	March
	MONTH.	date'2013-03-17'	3
	MONYY.	date'2013-03-17'	MAR13
	WEEKDATE.	date'2013-03-17'	Monday, March 17, 2013
	WEEKDATX.	date'2013-03-17'	Monday 17 March 2013
	WEEKDAY.	date'2013-03-17'	2
Datetime formats	DTDATE.	timestamp'2013-03-18 14:22:21'	17MAR13
	DTMONYY.	timestamp'2013-03-18 14:22:21'	MAR13
	DTWKDATX.	timestamp'2013-03-18 14:22:21'	Monday, March 17 2013
	DTYEAR.	timestamp'2013-03-18 14:22:21'	2013
	DTYYQC.	timestamp'2013-03-18 14:22:21'	13:1
Quarter formats	QTR.	date'2013-03-17'	1
	QTRR.	date'2013-03-17'	I
Time formats	HHMM.	time'14:22:21'	14:22
	HOUR.	time'14:22:21'	14
	TIME.	time'14:22:21'	14:22:21
	TIMEAMPM.	time'14:22:21'	2:22:21 PM
	TOD.	time'14:22:21	14:22:21

Type	Language Element	Input	Formatted Result
Year formats	YEAR.	date'2013-03-17'	2013
	YYMM.	date'2013-03-17'	2013M03
	YYMMC.	date'2013-03-17'	2013:03
	YYMMD.	date'2013-03-17'	2013-03
	YYMMN.	date'2013-03-17'	201303
	YYMMP.	date'2013-03-17'	2013.03
	YYMMS.	date'2013-03-17'	2013/03
	YYMMDD.	date'2013-03-17'	13-03-17
	YYMMDDDB.	date'2013-03-17'	13 03 17
	YYMMDDC.	date'2013-03-17'	13:03:17
	YYMMDDD.	date'2013-03-17'	13-03-17
	YYMMDDN.	date'2013-03-17'	20130317
	YYMMDDP.	date'2013-03-17'	13.03.17
	YYMMDDS.	date'2013-03-17'	13/03/17
	YYMON.	date'2013-03-17'	2013MAR
Year/Quarter formats	YYQ.	date'2013-03-17'	2013Q1
	YYQC.	date'2013-03-17'	2013:1
	YYQD.	date'2013-03-17'	2013-1
	YYQN.	date'2013-03-17'	20131
	YYQP.	date'2013-03-17'	2013.1
	YYQS.	date'2013-03-17'	2013/1
	YYQR.	date'2013-03-17'	2013QI
	YYQRC.	date'2013-03-17'	2013:I
	YYQRD.	date'2013-03-17'	2013-I

Type	Language Element	Input	Formatted Result
	YYQRN.	date'2013-03-17'	2013I
	YYQRP.	date'2013-03-17'	2013.I
	YYQRS.	date'2013-03-17'	2013/I

DICTIONARY Tables

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

The following table describes the DICTIONARY tables that are available. For a complete description of all the columns in the DICTIONARY tables, see [Appendix 5, “DICTIONARY Table Descriptions,”](#) on page 545.

Table 3.17 DICTIONARY Tables

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

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

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

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

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

```
select * from dictionary.tables;
```

Note: Some of the DICTIONARY tables (for example, STATISTICS and COLUMN_STATISTICS) rely on table driver support for their information. If the table driver does not support the underlying function call, no rows are returned.

FedSQL Pass-Through Facility

Overview

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

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

CONNECTION TO Component of the FROM Clause

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

The CONNECTION TO component has the following syntax:

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

Arguments

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

Example:

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

For more information, see the [“SELECT Statement” on page 386](#).

EXECUTE Statement

You should use the EXECUTE statement if the native SQL does not produce a result set, such as DML and DDL statements. In addition, the EXECUTE statement accepts native

SQL that produces a result set. For more information, see [“EXECUTE Statement” on page 381](#).

FedSQL Implicit Pass-Through Facility

Overview

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

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

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

How to Use FedSQL Implicit Pass-Through

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

Single Source FedSQL Implicit Pass-Through

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

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

Multiple Source FedSQL Implicit Pass-Through

FedSQL can perform IP on queries that include multiple data sources. This is accomplished by breaking the query into multiple queries and passing these individual queries to their respective drivers. In addition to the restrictions listed in [“Single Source](#)

[FedSQL Implicit Pass-Through](#) ” on page 60, multiple source IP has the following additional limitations:

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

Transactions in FedSQL

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

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

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

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

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

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

FedSQL Reserved Words

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

Note: You can use delimited identifiers for terms that might otherwise be a reserved word. For example, to use the term “char” other than for a character declaration, you would use it as the delimited identifier “char”. For more information, see [“Delimited Identifiers”](#) on page 18.

Table 3.18 FedSQL Reserved Words A - D

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

Table 3.19 FedSQL Reserved Words E - O

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

Table 3.20 FedSQL Reserved Words P - Z

P	REPLACE	STATEMENT	UNION
PARAMETER	RESET	STATIC	UNIQUE
PARTIAL	RESTART	STATISTICS	UNKNOWN
PARTITION	RESTRICT	STDDEV_POP	UNLISTEN
PASSWORD	RESULT	STDDEV_SAMP	UNNEST
PATH	RETURN	STDIN	UNTIL
PENDANT	RETURNS	STDOUT	UPDATE
PERCENT_RANK	REVOKE	STORAGE	USAGE
PERCENTILE_CONT	RIGHT	STRICT	USER
PERCENTILE_DESC	ROLLBACK	SUBMULTISET	USING
PLACING	ROLLUP	SUBSTRING	V
POSITION	ROW	SYMMETRIC	VACUUM
PRECISION	ROWS	SYSID	VALID
PREPARE	ROW_NUMBER	SYSTEM	VALIDATOR
PRESERVE	RULE	SYSTEM_USER	VALUE
PRIMARY	S	T	VALUES
PRIOR	SCHEMA	TABLE	VARCHAR
PRIVILEGES	SCOPE	TABLESAMPLE	VARYING
PROCEDURAL	SCROLL	TEMP	VAR_POP
PROCEDURE	SEARCH	TEMPLATE	VAR_SAMP
R	SECOND	TEMPORARY	VERBOSE
RANK	SECURITY	THEN	VERSION
READ	SELECT	TIME	VIEW
READS	SENSITIVE	TIMESTAMP	VOLATILE
REAL	SPECIFIC	TIMEZONE_HOUR	W
RECHECK	SPECIFICTYPE	TIMEZONE_MINUTE	WHEN
RECURSIVE	SEQUENCE	TO	WHENEVER
REF	SERIALIZABLE	TOAST	WHERE
REFERENCES	SESSION	TRAILING	WIDTH_BUCKET
REFERENCING	SESSION_USER	TRANSACTION	WINDOW
REGR_AVGX	SET	TRANSLATE	WITH
REGR_AVGY	SETOF	TRANSLATION	WITHIN
REGR_COUNT	SHARE	TREAT	WITHOUT
REGR_INTERCEPT	SHOW	TRIGGER	WORK
REGR_R2	SIMILAR	TRIM	WRITE
REGR_SLOPE	SIMPLE	TRUE	Y
REGR_SXX	SMALLINT	TRUNCATE	YEAR
REGR_SXY	SOME	TRUSTED	Z
REGR_SYY	SQLEXCEPTION	TYPE	ZONE
REINDEX	SQLSTATE	U	
RELATIVE	SQLWARNING	UESCAPE	
RENAME	STABLE	UNENCRYPTED	
	START		

Chapter 4

FedSQL Formats

Overview of Formats	69
General Format Syntax	69
Using Formats in FedSQL	70
How to Store, Change, Delete, and Use Stored Formats	70
How to Format Output with the PUT Function	71
Validation of FedSQL Formats	71
FedSQL Format Examples	71
Using a User-Defined Format	72
SAS Output Delivery System and FedSQL	73
Format Categories	73
NLS Formats Supported by FedSQL	73
Formats Supported with the PUT Function, by Category	76
Dictionary	81
\$BASE64Xw. Format	81
\$BINARYw. Format	82
\$CHARw. Format	83
\$HEXw. Format	83
\$OCTALw. Format	84
\$QUOTEw. Format	85
\$REVERJw. Format	87
\$REVERSw. Format	88
\$UPCASEw. Format	89
\$w. Format	89
BESTw. Format	90
BESTDw.p Format	92
BINARYw. Format	93
COMMAw.d Format	94
COMMAXw.d Format	95
Dw.p Format	96
DATEw. Format	98
DATEAMPWw.d Format	99
DATETIMEw.d Format	101
DAYw. Format	103
DDMMYYw. Format	104
DDMMYYxw. Format	105
DOLLARw.d Format	107

DOLLARXw.d Format	108
DOWNAMEw. Format	109
DTDATEw. Format	110
DTMONYYw. Format	111
DTWKDATXw. Format	113
DTYEARw. Format	114
DTYYQCw. Format	115
Ew. Format	116
EUROw.d Format	117
EUROXw.d Format	118
FLOATw.d Format	119
FRACTw. Format	120
HEXw. Format	121
HHMMw.d Format	122
HOURLw.d Format	124
IEEEw.d Format	125
JULIANw. Format	126
MMDDYYw. Format	127
MMDDYYxw. Format	129
MMSSw.d Format	130
MMYYw. Format	131
MMYYxw. Format	133
MONNAMEw. Format	134
MONTHw. Format	135
MONYYw. Format	136
NEGPARENw.d Format	137
NENGOW. Format	138
OCTALw. Format	140
PERCENTw.d Format	141
PERCENTNw.d Format	142
QTRw. Format	143
QTRRw. Format	144
ROMANw. Format	145
SIZEKw.d Format	145
SIZEKMGw.d Format	146
TIMEw.d Format	148
TIMEAMPMw.d Format	149
TODw.d Format	151
VAXRBw.d Format	152
w.d Format	153
WEEKDATEw. Format	154
WEEKDATXw. Format	155
WEEKDAYw. Format	157
YEARw. Format	158
YENw.d Format	159
YYMMw. Format	160
YYMMxw. Format	161
YYMMDDw. Format	163
YYMMDDxw. Format	164
YYMONw. Format	166
YYQw. Format	167
YYQxw. Format	168
YYQRw. Format	170
YYQRxw. Format	171
YYQZw. Format	173
Zw.d Format	174

Overview of Formats

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

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

General Format Syntax

FedSQL formats have the following syntax:

```
[ $ ]format [w ] . [d]
```

Arguments

\$

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

format

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

w

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

d

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

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

Formats always contain a period (.) as a part of the name. If you omit the *w* and the *d* values from the format, SAS uses default values. The *d* value that you specify with a format tells FedSQL to display that many decimal places, regardless of how many decimal places are in the data. Formats never change or truncate the internally stored data values.

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

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

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

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

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

returns a blank value.

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

Using Formats in FedSQL

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

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

You specify formats in the CREATE TABLE statement as an attribute of the HAVING clause. For more information, see [“CREATE TABLE Statement” on page 359](#). For example, in the following statement, the column *profit* is declared with the EURO13.2 format.

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

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

FedSQL supports stored formats as follows:

- Both user-defined formats and formats that are supplied by SAS can be stored. For more information, see [“Using a User-Defined Format” on page 72](#).
- All formats that are supplied by SAS can be stored. For a list of formats, see *SAS Formats and Informats: Reference*. FedSQL does not validate the formats. If the stored format is invalid, an error occurs, but only when the invalid format is used in the client application.
- Formats can be stored only for columns of data types CHAR and DOUBLE.
- To access the stored formats, you must have a Base SAS session available. The Base SAS session contains the SAS format definitions and SAS catalog file that stores the user-defined SAS format definitions.
- You can store and retrieve format names. The format name is associated with a column by storing the format as a metadata attribute on the column. The metadata then can be retrieved for subsequent operations.

How to Format Output with the PUT Function

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

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

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

- If the PUT function is used without a format, an error occurs.
- The PUT function supports a subset of the formats that are available for Base SAS when the FedSQL language is executed outside a Base SAS session. For a list, see [“Formats Supported with the PUT Function, by Category” on page 76](#).
- Formats can be associated with any of the data types that are supported by FedSQL. However, the data types are converted. Any value that is passed to the PUT function with a numeric format is converted to NVARCHAR, VARBINARY, or BINARY. The type conversions are carried out based on the format name. Any value that is passed with a character format to the PUT function is converted to NVARCHAR.
- The format that is specified in PUT is transient. The PUT function does not affect the stored data.
- The PUT function does not require a Base SAS session to be available; however, the functionality is limited when a session is not available.
- The PUT function cannot be used on SASHDAT files. FedSQL support for SASHDAT files is Write-only.

See also the [“PUT Function” on page 286](#).

Validation of FedSQL Formats

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

The PUT function validates the specified format upon use.

FedSQL Format Examples

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

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

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

Using a User-Defined Format

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

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

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

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

The content of the source Employees table is shown in [Display 4.1 on page 72](#). The output of the PUT function is shown in [Display 4.2 on page 72](#).

Display 4.1 Content of the Source EMPLOYEES Table

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

Display 4.2 Content of the Employees Table After the PUT Function Is Applied

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

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

SAS Output Delivery System and FedSQL

The SAS Output Delivery System (ODS), which PROC FEDSQL uses to display results, by default rounds numeric output to appear inside 8 spaces. To display numeric output with the full precision of which FedSQL is capable, use the PUT function with the BEST16. format, as follows:

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

Format Categories

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

Character

writes character data values from character variables.

Date and Time

writes character data values from character variables.

Numeric

writes numeric data values from numeric variables.

NLS Formats Supported by FedSQL

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

Category	Language Element	Description
Date and Time	NLDATEw.	Converts a SAS date value to the date value of the specified locale, and then writes the date value as a date.
	NLDATEMDw.	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.

Category	Language Element	Description
	NLDATEMN _w .	Converts a SAS date value to the date value of the specified locale, and then writes the value as the name of the month.
	NLDATEW _w .	Converts a SAS date value to the date value of the specified locale, and then writes the value as the date and the day of the week.
	NLDATEWN _w .	Converts the SAS date value to the date value of the specified locale, and then writes the date value as the day of the week.
	NLDATEYM _w .	Converts the SAS date value to the date value of the specified locale, and then writes the date value as the year and the name of the month.
	NLDATEYQ _w .	Converts the SAS date value to the date value of the specified locale, and then writes the date value as the year and the quarter.
	NLDATEYR _w .	Converts the SAS date value to the date value of the specified locale, and then writes the date value as the year.
	NLDATEYW _w .	Converts the SAS date value to the date value of the specified locale, and then writes the date value as the year and the week.
	NLDATMAP _w .	Converts a SAS datetime value to the datetime value of the specified locale, and then writes the value as a datetime with a.m. or p.m.
	NLDATMDT _w .	Converts the SAS datetime value to the datetime value of the specified locale, and then writes the value as the name of the month, day of the month, and year.
	NLDATMMD _w .	Converts the SAS datetime value to the datetime value of the specified locale, and then writes the value as the name of the month and the day of the month.
	NLDATMMN _w .	Converts the SAS datetime value to the datetime value of the specified locale, and then writes the value as the name of the month.
	NLDATMTM _w .	Converts the time portion of a SAS datetime value to the time-of-day value of the specified locale, and then writes the value as a time of day.
	NLDATM _w .	Converts a SAS datetime value to the datetime value of the specified locale, and then writes the value as a datetime.

Category	Language Element	Description
	NLDATMW _w .	Converts SAS datetime values to the locale sensitive datetime string as the day of the week and the datetime.
	NLDATMWN _w .	Converts a SAS datetime value to the datetime value of the specified locale, and then writes the value as the day of the week.
	NLDATMYM _w .	Converts the SAS datetime value to the datetime value of the specified locale, and then writes the value as the year and the name of the month.
	NLDATMYQ _w .	Converts the SAS datetime value to the datetime value of the specified locale, and then writes the value as the year and the quarter of the year.
	NLDATMYR _w .	Converts the SAS datetime value to the datetime value of the specified locale, and then writes the value as the year.
	NLDATMYW _w .	Converts the SAS datetime value to the datetime value of the specified locale, and then writes the value as the year and the name of the week.
	NLTIMAP _w .	Converts a SAS time value to the time value of a specified locale, and then writes the value as a time value with a.m. or p.m. NLTIMAP also converts SAS date-time values.
	NLTIME _w .	Converts a SAS time value to the time value of the specified locale, and then writes the value as a time value. NLTIME also converts SAS date-time values.
Numeric	NLBEST _w .	Writes the best numerical notation based on the locale.
	NLMNY _{w.d}	Writes the monetary format of the local expression in the specified locale using local currency.
	NLMNYI _{w.d}	Writes the monetary format of the international expression in the specified locale.
	NLNUM _{w.d}	Writes the numeric format of the local expression in the specified locale.
	NLNUMI _{w.d}	Writes the numeric format of the international expression in the specified locale.
	NLPCT _{w.d}	Writes percentage data of the local expression in the specified locale.

Category	Language Element	Description
	NLPCTI _{w.d}	Writes percentage data of the international expression in the specified locale.
	NLPCTN _{w.d}	Produces percentages, using a minus sign for negative values.
	NLPCTP _{w.d}	Writes locale-specific numeric values as percentages. Writes locale-specific numeric values as percentages.
	NLPVALUE _{w.d}	Writes p-values of the local expression in the specified locale.
	NLSTRMON _{w.d}	Writes the month name in the specified locale.
	NLSTRQTR _{w.d}	Writes a numeric value as the quarter-of-the-year in the specified locale.
	NLSTRWK _{w.d}	Writes a numeric value as the day-of-the-week in the specified locale.

Formats Supported with the PUT Function, by Category

Category	Language Elements	Description
Character	\$BASE64X _w . Format (p. 81)	Converts character data into ASCII text by using Base 64 encoding.
	\$BINARY _w . Format (p. 82)	Converts character data to binary representation.
	\$CHAR _w . Format (p. 83)	Writes standard character data.
	\$HEX _w . Format (p. 83)	Converts character data to hexadecimal representation.
	\$OCTAL _w . Format (p. 84)	Converts character data to octal representation.
	\$QUOTE _w . Format (p. 85)	Writes data values that are enclosed in single quotation marks.
	\$REVERJ _w . Format (p. 87)	Writes character data in reverse order and preserves blanks.
	\$REVERSw. Format (p. 88)	Writes character data in reverse order and left aligns.
	\$UPCASE _w . Format (p. 89)	Converts character data to uppercase.
	\$ _w . Format (p. 89)	Writes standard character data.

Category	Language Elements	Description
Date and Time	DATE _w . Format (p. 98)	Writes SAS date values in the form ddmmyy, ddmmyyyy, or dd-mmm-yyyy.
	DATEAMP _{Mw.d} Format (p. 99)	Writes SAS datetime values in the form ddmmyy:hh:mm:ss.ss with AM or PM.
	DATETIME _{w.d} Format (p. 101)	Writes SAS datetime values in the form ddmmyy:hh:mm:ss.ss.
	DAY _w . Format (p. 103)	Writes SAS date values as the day of the month.
	DDMMYY _w . Format (p. 104)	Writes SAS date values in the form ddmm[yy]yy or dd/mm/[yy]yy, where a forward slash is the separator and the year appears as either 2 or 4 digits.
	DDMMYY _{xw} . Format (p. 105)	Writes SAS date values in the form ddmm[yy]yy or ddXmmX[yy]yy, where X represents a specified separator and the year appears as either 2 or 4 digits.
	DOWNAME _w . Format (p. 109)	Writes SAS date values as the name of the day of the week.
	DTDATE _w . Format (p. 110)	Expects a SAS datetime value as input and writes the SAS date values in the form ddmmyy or ddmmyyyy.
	DTMONYY _w . Format (p. 111)	Writes the date part of a SAS datetime value as the month and year in the form mmyy or mmyyyy.
	DTWKDATX _w . Format (p. 113)	Writes the date part of a SAS datetime value as the day of the week and the date in the form day-of-week, dd month-name yy (or yyyy).
	DTYEAR _w . Format (p. 114)	Writes the date part of a SAS datetime value as the year in the form yy or yyyy.
	DTYYQC _w . Format (p. 115)	Writes the date part of a SAS datetime value as the year and the quarter, and separates them with a colon (:).
	HHMM _{w.d} Format (p. 122)	Writes SAS time values as hours and minutes in the form hh:mm.
	HOUR _{w.d} Format (p. 124)	Writes SAS time values as hours and decimal fractions of hours.
	JULIAN _w . Format (p. 126)	Writes SAS date values as Julian dates in the form yyddd or yyyyddd.
	MMDDYY _w . Format (p. 127)	Writes SAS date values in the form mmdd[yy]yy or mm/dd/[yy]yy, where a forward slash is the separator and the year appears as either 2 or 4 digits.
	MMDDYY _{xw} . Format (p. 129)	Writes SAS date values in the form mmdd[yy]yy or mmXddX[yy]yy, where X represents a specified separator and the year appears as either 2 or 4 digits.

Category	Language Elements	Description
	MMSS _{w.d} Format (p. 130)	Writes SAS time values as the number of minutes and seconds since midnight.
	MMYY _{w.} Format (p. 131)	Writes SAS date values in the form mmM[yy]yy, where M is the separator and the year appears as either 2 or 4 digits.
	MMYY _{xw.} Format (p. 133)	Writes SAS date values in the form mm[yy]yy or mmX[yy]yy. The x in the format name is a character that represents the special character. The special character separates the month and the year. That special character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits.
	MONNAME _{w.} Format (p. 134)	Writes SAS date values as the name of the month.
	MONTH _{w.} Format (p. 135)	Writes SAS date values as the month of the year.
	MONYY _{w.} Format (p. 136)	Writes SAS date values as the month and the year in the form mmmmyy or mmmmyyyy.
	NENGO _{w.} Format (p. 138)	Writes SAS date values as Japanese dates in the form e.yymmdd.
	QTR _{w.} Format (p. 143)	Writes SAS date values as the quarter of the year.
	QTRR _{w.} Format (p. 144)	Writes SAS date values as the quarter of the year in Roman numerals.
	TIME _{w.d} Format (p. 148)	Writes SAS time values as hours, minutes, and seconds in the form hh:mm:ss.ss using the military 24-hour clock.
	TIMEAMP _{w.d} Format (p. 149)	Writes SAS time values as hours, minutes, and seconds in the form hh:mm:ss.ss with AM or PM.
	TOD _{w.d} Format (p. 151)	Writes SAS time values and the time portion of SAS datetime values in the form hh:mm:ss.ss.
	WEEKDATE _{w.} Format (p. 154)	Writes SAS date values as the day of the week and the date in the form day-of-week, month-name dd, yy (or yyyy).
	WEEKDATX _{w.} Format (p. 155)	Writes SAS date values as the day of the week and date in the form day-of-week, dd month-name yy (or yyyy).
	WEEKDAY _{w.} Format (p. 157)	Writes SAS date values as the day of the week.
	YEAR _{w.} Format (p. 158)	Writes SAS date values as the year.
	YYMM _{w.} Format (p. 160)	Writes SAS date values in the form [yy]yyMmm, where M is the separator and the year appears as either 2 or 4 digits.
	YYMM _{xw.} Format (p. 161)	Writes SAS date values in the form [yy]yyymm or [yy]yy-mm. The x in the format name represents the special character that separates the year and the month. This special character can be a hyphen (-), period (.), slash(/), colon(:), or no separator. The year can be either 2 or 4 digits.

Category	Language Elements	Description
	YYMMDDw. Format (p. 163)	Writes SAS date values in the form yymmdd or [yy]yy-mm-dd, where a hyphen (-) is the separator and the year appears as either 2 or 4 digits.
	YYMMDDxw. Format (p. 164)	Writes date values in the form [yy]yymmdd or [yy]yy-mm-dd. The x in the format name is a character that represents the special character which separates the year, month, and day. This special character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits.
	YYMONw. Format (p. 166)	Writes SAS date values in the form yymmm or yyyyymm.
	YYQw. Format (p. 167)	Writes SAS date values in the form [yy]yyQq, where Q is the separator, the year appears as either 2 or 4 digits, and q is the quarter of the year.
	YYQxw. Format (p. 168)	Writes SAS date values in the form [yy]yyq or [yy]yy-q. The x in the format name is a character that represents the special character that separates the year and the quarter of the year. This character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits.
	YYQRw. Format (p. 170)	Writes SAS date values in the form [yy]yyQqr, where Q is the separator, the year appears as either 2 or 4 digits, and qr is the quarter of the year expressed in roman numerals.
	YYQRxw. Format (p. 171)	Writes date values in the form [yy]yyqr or [yy]yy-qr. The x in the format name is a character that represents the special character that separates the year and the quarter of the year. This character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits and qr is the quarter of the year in roman numerals.
	YYQZw. Format (p. 173)	Writes SAS date values in the form [yy] yyqq. The year appears as 2 or 4 digits, and qq is the quarter of the year.
Numeric	BESTw. Format (p. 90)	SAS chooses the best notation.
	BESTDw.p Format (p. 92)	Prints numeric values, lining up decimal places for values of similar magnitude, and prints integers without decimals.
	BINARYw. Format (p. 93)	Converts numeric values to binary representation.
	COMMAw.d Format (p. 94)	Writes numeric values with a comma that separates every three digits and a period that separates the decimal fraction.
	COMMAXw.d Format (p. 95)	Writes numeric values with a period that separates every three digits and a comma that separates the decimal fraction.
	Dw.p Format (p. 96)	Prints variables, possibly with a great range of values, lining up decimal places for values of similar magnitude.
	DOLLARw.d Format (p. 107)	Writes numeric values with a leading dollar sign, a comma that separates every three digits, and a period that separates the decimal fraction.

Category	Language Elements	Description
	DOLLARX _{w,d} Format (p. 108)	Writes numeric values with a leading dollar sign, a period that separates every three digits, and a comma that separates the decimal fraction.
	E _w . Format (p. 116)	Writes numeric values in scientific notation.
	EURO _{w,d} Format (p. 117)	Writes numeric values with a leading euro symbol (€), a comma that separates every three digits, and a period that separates the decimal fraction.
	EUROX _{w,d} Format (p. 118)	Writes numeric values with a leading euro symbol (€), a period that separates every three digits, and a comma that separates the decimal fraction.
	FLOAT _{w,d} Format (p. 119)	Generates a native single-precision, floating-point value by multiplying a number by 10 raised to the dth power.
	FRACT _w . Format (p. 120)	Converts numeric values to fractions.
	HEX _w . Format (p. 121)	Converts real binary (floating-point) values to hexadecimal representation.
	IEEE _{w,d} Format (p. 125)	Generates an IEEE floating-point value by multiplying a number by 10 raised to the dth power.
	NEGPAREN _{w,d} Format (p. 137)	Writes negative numeric values in parentheses.
	OCTAL _w . Format (p. 140)	Converts numeric values to octal representation.
	PERCENT _{w,d} Format (p. 141)	Writes numeric values as percentages.
	PERCENTN _{w,d} Format (p. 142)	Produces percentages, using a minus sign for negative values.
	ROMAN _w . Format (p. 145)	Writes numeric values as roman numerals.
	SIZEK _{w,d} Format (p. 145)	Writes a numeric value in the form nK for kilobytes.
	SIZEKMG _{w,d} Format (p. 146)	Writes a numeric value in the form nKB for kilobytes, nMB for megabytes, or nGB for gigabytes.
	VAXRB _{w,d} Format (p. 152)	Writes real binary (floating-point) data in VMS format.
	_{w,d} Format (p. 153)	Writes standard numeric data one digit per byte.
	YEN _{w,d} Format (p. 159)	Writes numeric values with yen signs, commas, and decimal points.
	Z _{w,d} Format (p. 174)	Writes standard numeric data with leading 0s.

Dictionary

\$BASE64Xw. Format

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

Category: Character

Alignment: Left

Syntax

\$BASE64X^w.

Arguments

^w
specifies the width of the output field.

Default 1

Range 1–32767

Details

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

The following are some uses of Base 64 encoding:

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

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

Example

Statements	Results
<pre>select put ("FCA01A7993BC", \$base64x64.);</pre>	RkNBMDFBNzk5M0JD
<pre>select put ("MyPassword", \$base64x64.);</pre>	TXlQYXNzd29yZA==

Statements	Results
<pre>select put ("www.mydomain.com/myhiddenURL", d3d3Lm15ZG9tYWluLmNvbi9teWhpZGRlblVSTA== \$base64x64.);</pre>	

\$BINARYw. Format

Converts character data to binary representation.

Category: Character

Alignment: Left

Syntax

\$BINARY*w*.

Arguments

w
specifies the width of the output field.

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

Range 1–32767

Comparisons

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

Example

Statements	Results	
	ASCII	EBCDIC
	-----1-----2	-----1-----2
<pre>select put('AB', \$binary16.);</pre>	0100000101000010	1100000111000010

See Also

Formats:

- “BINARY*w*. Format” on page 93
- “HEX*w*. Format” on page 83

\$CHARw. Format

Writes standard character data.

Category: Character
Alignment: Left
Alias: \$w.

Syntax

\$CHARw.

Arguments

w
 specifies the width of the output field.

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

Range 1–32767

Comparisons

The \$CHARw. and \$w. formats are identical, and they do not trim leading blanks.

Example

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

See Also

Formats:

- “\$w. Format” on page 89

\$HEXw. Format

Converts character data to hexadecimal representation.

Category: Character
Alignment: Left

Syntax

\$HEX*w*.

Arguments

w	specifies the width of the output field.
Default	The default width is calculated based on the length of the variable to be printed.
Range	1–32767
Tips	<p>To ensure that SAS writes the full hexadecimal equivalent of your data, make <i>w</i> twice the length of the variable or field that you want to represent.</p> <p>If <i>w</i> is greater than twice the length of the variable that you want to represent, \$HEX<i>w</i>. pads it with blanks.</p>

Details

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

Comparisons

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

Example

Statements	Results
<pre>select put ('AB',\$hex5.);</pre>	<pre>-----1-----2 4142</pre>

See Also

- Formats:**
- “\$BINARY*w*. Format” on page 82
 - “HEX*w*. Format” on page 121

\$OCTAL*w*. Format

Converts character data to octal representation.

Category: Character

Alignment: Left

Syntax

\$OCTALw.

Arguments

w	specifies the width of the output field.
Default	The default width is calculated based on the length of the variable to be printed.
Range	1–32767
Tip	Because each character value generates three octal characters, increase the value of w by three times the length of the character value.

Comparisons

The \$OCTALw. format converts character values to the octal representation of their character codes. The OCTALw. format converts numeric values to octal representation.

Example

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

See Also

Formats:

- [“OCTALw. Format” on page 140](#)

\$QUOTEw. Format

Writes data values that are enclosed in single quotation marks.

Category:	Character
Alignment:	Left

Syntax

\$QUOTEw.

Arguments

w

specifies the width of the output field.

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

Range 2–32767

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

Details

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

The following list describes the output that SAS produces when you use the \$QUOTE*w*. format.

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

Example

Statements	Results
	-----1-----2
<code>select put('SAS',\$quote.);</code>	"SAS"
<code>select put('SAS''s',\$quote.);</code>	"SAS's"
<code>select put('ad'verb',\$quote16.);</code>	"ad'verb"
<code>select put('"ad" verb',\$quote16.);</code>	""ad""verb""
<code>select put('"ad"'verb',\$quote20.);</code>	""ad""'verb""

\$REVERJw. Format

Writes character data in reverse order and preserves blanks.

Category: Character

Alignment: Right

Syntax

\$REVERJ*w*.

Arguments

w
specifies the width of the output field.

Default 1 if *w* is not specified

Range 1–32767

Comparisons

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

Example

Statements*	Results*
	-----1
<code>select put('ABCD###',\$reverj7.);</code>	###DCBA

Statements*	Results*
select put('###ABCD',\$reverj7.);	DCBA###
* The character # represents a blank space.	

See Also

Formats:

- [“\\$REVERSw. Format” on page 88](#)

\$REVERSw. Format

Writes character data in reverse order and left aligns.

Category:	Character
Alignment:	Left

Syntax

\$REVERSw.

Arguments

<i>w</i>	specifies the width of the output field.
Default	1 if <i>w</i> is not specified
Range	1–32767

Comparisons

The \$REVERSw. format is similar to the \$REVERJw. format except that \$REVERJw. does not left align the result.

Example

Statements*	Results
	-----1
select put('ABCD###',\$revers7);	DCBA
select put('###ABCD',\$revers7.);	DCBA
* The character # represents a blank space.	

See Also

Formats:

- [“\\$REVERJw. Format” on page 87](#)

\$UPCASEw. Format

Converts character data to uppercase.

Category: Character

Alignment: Left

Syntax

\$UPCASEw.

Arguments

w

specifies the width of the output field.

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

Range 1–32767

Details

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

Example

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

\$w. Format

Writes standard character data.

Category: Character

Alignment: Left

Alias: \$Fw.

Syntax

`$w.`

Arguments

w	specifies the width of the output field.
Default	1 if the length of the variable is undefined; otherwise, the length of the variable.
Range	1–32767

Comparisons

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

Example

Statements*	Results*
	-----1-----2
<code>select put('#Cary',\$5.);</code>	#Cary
<code>select put('#Cary',\$f5.);</code>	#Cary
<code>select put(Carolina,\$5.);</code>	Carol

* The character # represents a blank space.

See Also

Formats:

- “\$CHARw. Format” on page 83
- “w.d Format” on page 153

BESTw. Format

SAS chooses the best notation.

Category:	Numeric
Alignment:	Right

Syntax

`BESTw.`

Arguments

w

specifies the width of the output field.

Default 12

Range 1–32

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

Details

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

Comparisons

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

Example

Statements	Results
	-----1-----2
select put(1257000,best6.);	1.26E6
select put(1257000,best3.);	1E6

See Also

Formats:

- [“BESTDw.p Format” on page 92](#)
- [“Dw.p Format” on page 96](#)
- [“w.d Format” on page 153](#)

BESTDw.p Format

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

Category: Numeric

Alignment: Right

Syntax

BESTDw.*p*

Arguments

w
specifies the width of the output field.

Default	12
Range	1–32

p
specifies the precision.

Default	3
Range	0 to <i>w</i> –1
Requirement	must be less than <i>w</i>
Tip	If <i>p</i> is omitted or is specified as 0, then <i>p</i> is set to 3.

Details

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

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

Comparisons

- The BESTw. format writes as many significant digits as possible in the output field, but if the numbers vary in magnitude, the decimal points do not line up. Integers print without a decimal.
- The Dw.p format writes numbers with the desired precision and more alignment than the BESTw format.
- The BESTDw.p format is a combination of the BESTw. format and the Dw.p format in that it formats all numeric data, and it does a better job of aligning decimals than the BESTw. format.

- The *w.d* format aligns decimal points, if possible, but it does not necessarily show the same precision for all numbers.

Example

Statements	Results
	-----1-----
<code>select put (12345, bestd14.);</code>	12345
<code>select put (123.45, bestd14.);</code>	123.4500000
<code>select put (1.2345, bestd14.);</code>	1.2345000
<code>select put (.12345, bestd14.);</code>	0.1234500
<code>select put (1.23456789, bestd14.);</code>	1.23456789

See Also

Formats:

- [“BESTw. Format” on page 90](#)
- [“Dw.p Format” on page 96](#)
- [“w.d Format” on page 153](#)

BINARYw. Format

Converts numeric values to binary representation.

Category: Numeric

Alignment: Left

Syntax

BINARYw.

Arguments

w
specifies the width of the output field.

Default 8

Range 1–64

Comparisons

BINARY_w. converts numeric values to binary representation. The \$BINARY_w. format converts character values to binary representation.

Example

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

See Also

Formats:

- [“\\$BINARY_w. Format” on page 82](#)
- [“HEX_w. Format” on page 121](#)

COMMA_{w,d} Format

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

Category:	Numeric
Alignment:	Right

Syntax

COMMA_w.^[*d*]

Arguments

<i>w</i>	specifies the width of the output field.
Default	6
Range	1–32
Tip	Make <i>w</i> wide enough to write the numeric values, the commas, and the optional decimal point.
<i>d</i>	specifies the number of digits to the right of the decimal point in the numeric value.

Range	0–31
Requirement	must be less than <i>w</i>

Comparisons

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

Example

Statements	Results
	-----1-----2
<code>select put (23451.23,comma10.2);</code>	23,451.23
<code>select put (123451.234,comma10.2);</code>	123,451.23

See Also

Formats:

- [“COMMAXw.d Format” on page 95](#)
- [“DOLLARw.d Format” on page 107](#)

COMMAXw.d Format

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

Category:	Numeric
Alignment:	Right

Syntax

COMMAX[w].d

Arguments

w
specifies the width of the output field.

Default 6

Range 1–32

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

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

Range 0–31

Requirement must be less than *w*

Comparisons

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

Example

Statements	Results
	-----1-----2
select put (23451.23,commax10.2);	23.451,23
select put (123451.234,commax10.2);	123.451,23

See Also

- Formats:
- [“COMMA*w.d* Format” on page 94](#)
 - [“DOLLARX*w.d* Format” on page 108](#)

Dw.p Format

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

Category: Numeric

Alignment: Right

Syntax

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

Arguments

w specifies the width of the output field.

Default	12
Range	1–32

p

specifies the significant digits.

Default	3
Range	0–16
Requirement	must be less than <i>w</i>

Details

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

Comparisons

- The *BESTw.* format writes as many significant digits as possible in the output field, but if the numbers vary in magnitude, the decimal points do not line up.
- *Dw.p* writes numbers with the desired precision and more alignment than *BESTw.*
- The *BESTDw.p* format is a combination of the *BESTw.* format and the *Dw.p* format in that it formats all numeric data, and it does a better job of aligning decimals than the *BESTw.* format.
- The *w.d* format aligns decimal points, if possible, but it does not necessarily show the same precision for all numbers.

Example

Statements	Results
	-----1-----2
<code>select put (12345,d10.4);</code>	12345.0
<code>select put (1234.5,d10.4);</code>	1234.5
<code>select put (123.45,d10.4);</code>	123.45000
<code>select put (12.345,d10.4);</code>	12.34500
<code>select put (1.2345,d10.4);</code>	1.23450
<code>select put (.12345,d10.4);</code>	0.12345

See Also

Formats:

- “BESTw. Format” on page 90
- “BESTDw.p Format” on page 92
- “w.d Format” on page 153

DATEw. Format

Writes SAS date values in the form *ddmmyy*, *ddmmyyyy*, or *dd-mmm-yyyy*.

Category: Date and Time

Alignment: Right

Syntax

DATE*w.*

Arguments

w

specifies the width of the output field.

Default 7

Range 5–9

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

Details

The DATEw. format writes SAS date values in the form *ddmmyy*, *ddmmyyyy*, or *dd-mmm-yyyy* where

dd

is an integer that represents the day of the month.

mmm

is the first three letters of the month name.

yy or *yyyy*

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

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

Example

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

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

See Also

Formats:

- [“DATETIMEw.d Format” on page 101](#)
- [“DAYw. Format” on page 103](#)
- [“DDMMYYw. Format” on page 104](#)
- [“DTDATEw. Format” on page 110](#)
- [“JULIANw. Format” on page 126](#)
- [“MMDDYYw. Format” on page 127](#)
- [“MONTHw. Format” on page 135](#)
- [“NENGOW. Format” on page 138](#)
- [“WEEKDATEw. Format” on page 154](#)
- [“YEARw. Format” on page 158](#)

Functions:

- [“CURRENT_DATE Function” on page 223](#)
- [“DATEPART Function” on page 230](#)
- [“DAY Function” on page 231](#)
- [“MAKEDATE Function” on page 260](#)

DATEAMPMw.d Format

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

Category: Date and Time

Alignment: Right

Syntax

DATEAMPMW.*d*

Arguments

<i>w</i>	specifies the width of the output field.
Default	19
Range	7–40
Tip	SAS requires a minimum <i>w</i> value of 13 to write AM or PM. For widths between 10 and 12, SAS writes a 24-hour clock time.
<i>d</i>	specifies the number of digits to the right of the decimal point in the seconds value.
Range	0–39
Requirement	must be less than <i>w</i>
Note	If $w - d < 17$, SAS truncates the decimal values.

Details

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

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

mmm
is the first three letters of the month name.

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

hh
is an integer that represents the hour.

mm
is an integer that represents the minutes.

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

Comparisons

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

Example

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

Statements	Results
	-----1-----2-----
select put (1679344494,dateampm.);	19MAR13:08:34:54 PM
select put (1679344494,dateampm7.);	19MAR13
select put (1679344494,dateampm10.);	19MAR13:20
select put (1679344494,dateampm13.);	19MAR13:08 PM
select put (1679344494,dateampm22.2.);	19MAR13:08:34:54.00 PM

See Also

Formats:

- [“DATETIMEw.d Format” on page 101](#)
- [“TIMEAMPMw.d Format” on page 149](#)

DATETIMEw.d Format

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

Category: Date and Time

Alignment: Right

Syntax

DATETIMEw.[\[d\]](#)

Arguments

w

specifies the width of the output field.

Default 16

Range 7–40

Tips

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

If $w - d < 17$, SAS truncates the decimal values.

d

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

Range 0–39

Requirement must be less than *w*

Details

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

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

mmm
is the first three letters of the month name.

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

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

mm
is an integer that represents the minutes.

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

Comparisons

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

Example

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

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

See Also

Formats:

- [“DATEAMPMw.d Format” on page 99](#)
- [“DDMMYYxw. Format” on page 105](#)
- [“MMDDYYw. Format” on page 127](#)

DAYw. Format

Writes SAS date values as the day of the month.

Category: Date and Time

Alignment: Right

Syntax

DAY*w.*

Arguments

w
specifies the width of the output field.

Default 2

Range 2–32

Example

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

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

See Also

Formats:

- [“DATEw. Format” on page 98](#)

Functions:

- [“DAY Function” on page 231](#)

DDMMYYw. Format

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

Category: Date and Time

Alignment: Right

Syntax

DDMMYYw.

Arguments

w	specifies the width of the output field.
Default	8
Range	2–10
Interaction	When <i>w</i> has a value of from 2 to 5, the date appears with as much of the day and the month as possible. When <i>w</i> is 7, the date appears as a two-digit year without slashes.

Details

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

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

/
is the separator.

mm
is an integer that represents the month.

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

Example

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

Statements	Results
	-----1-----
select put(19704,ddmmyy.);	12/12/13

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

See Also

Formats:

- [“DATEw. Format” on page 98](#)
- [“DDMMYYxw. Format” on page 105](#)
- [“MMDDYYw. Format” on page 127](#)
- [“YYMMDDw. Format” on page 163](#)

Functions:

- [“DAY Function” on page 231](#)
- [“MONTH Function” on page 275](#)
- [“YEAR Function” on page 318](#)

DDMMYYxw. Format

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

Category: Date and Time

Alignment: Right

Syntax

DDMMYY*xw*.

Arguments

x

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

B

separates with a blank

C

separates with a colon

- D separates with a hyphen
 - N indicates no separator
 - P separates with a period
 - S separates with a slash.
- w* specifies the width of the output field.

Default	8
Range	2–10
Interactions	When <i>w</i> has a value of from 2 to 5, the date appears with as much of the day and the month as possible. When <i>w</i> is 7, the date appears as a two-digit year without separators.
	When <i>x</i> has a value of N, the width range changes to 2–8.

Details

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

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

X
is a specified separator.

mm
is an integer that represents the month.

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

Example

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

Statements	Results
	----+----1----+
select put(19431,ddmmyyc5.);	14:03
select put(19431,ddmmyyd8.);	14-03-13
select put(19431,ddmmyyn8.);	14032013
select put(19431,ddmmyyp10.);	14.03.2013

See Also

Formats:

- “DATEw. Format” on page 98
- “DDMMYYw. Format” on page 104
- “MMDDYYxw. Format” on page 129
- “YYMMDDxw. Format” on page 164

Functions:

- “DAY Function” on page 231
- “MONTH Function” on page 275
- “YEAR Function” on page 318

DOLLARw.d Format

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

Category: Numeric

Alignment: Right

Syntax

DOLLARw.[d]

Arguments

w

specifies the width of the output field.

Default 6

Range 2–32

d

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

Range 0–31

Requirement must be less than w

Details

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

The hexadecimal representation of the code for the dollar sign character (\$) is 5B on EBCDIC systems and 24 on ASCII systems. The monetary character that these codes

represent might be different in other countries, but DOLLAR $w.d$ always produces one of these codes.

Comparisons

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

Example

Statements	Results
	-----1-----
<code>select put(1254.71,dollar10.2);</code>	...\$1,254.71

See Also

Formats:

- [“COMMA \$w.d\$ Format” on page 94](#)
- [“DOLLARX \$w.d\$ Format” on page 108](#)
- [“EURO \$w.d\$ Format” on page 117](#)

DOLLARX $w.d$ Format

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

Category: Numeric

Alignment: Right

Syntax

DOLLARX $w.[d]$

Arguments

w specifies the width of the output field.

Default 6

Range 2–32

d

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

Default	0
Range	2–31
Requirement	must be less than <i>w</i>

Details

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

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

Comparisons

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

Example

Statements	Results
	-----1-----+
<pre>select put(1254.71,dollarx10.2);</pre>	\$1.254,71

See Also

Formats:

- [“COMMAXw.d Format” on page 95](#)
- [“DOLLARw.d Format” on page 107](#)

DOWNAMEw. Format

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

Category: Date and Time

Alignment: Right

Syntax

DOWNNAME^w.

Arguments

w	specifies the width of the output field.
Default	9
Range	1–32
Tip	If you omit <i>w</i> , SAS prints the entire name of the day.

Example

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

Statements	Results
	-----1
<pre>select put(19431,downname.);</pre>	Monday

See Also

Formats:

- “DATE^w. Format” on page 98
- “DTWKDATX^w. Format” on page 113
- “WEEKDATE^w. Format” on page 154
- “WEEKDATX^w. Format” on page 155
- “WEEKDAY^w. Format” on page 157

DTDATE^w. Format

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

Category: Date and Time

Alignment: Right

Syntax

DTDATE^w.

Arguments

w	specifies the width of the output field.
Default	7
Range	5–9
Tip	Use a width of 9 to print a 4-digit year.

Details

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

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

mmm
are the first three letters of the month name.

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

Comparisons

The DTDATEx. format produces the same type of output that the DATEx. format produces. The difference is that the DTDATEx. format requires a SAS datetime value.

Example

The example table uses the input value of 1510285759, which is the SAS datetime value that corresponds to 3:49:19 AM on November 10, 2013, and prints both a two-digit and a four-digit year for the DTDATEx. format.

Statements	Results
	-----1
<pre>select put(1510285759,dtdate.);</pre>	10NOV13
<pre>select put(1510285759,dtdate9.);</pre>	10NOV2013

See Also

Formats:

- [“DATEw. Format” on page 98](#)

DTMONYYw. Format

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

Category: Date and Time

Alignment: Right

Syntax

DTMONYY^w.

Arguments

^w
specifies the width of the output field.

Default	5
Range	5–7

Details

The DTMONYY^w. format writes SAS datetime values in the form *mmm*yy or *mmm*yyyy, where

mmm
is the first three letters of the month name.

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

Comparisons

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

Example

The example table uses as input the value 1678898894, which is the SAS datetime value that corresponds to March 14, 2013, at 04:48:14 PM.

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

See Also

Formats:

- “DATETIMEw.d Format” on page 101
- “MONYYw. Format” on page 136

DTWKDATXw. Format

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

Category: Date and Time

Alignment: Right

Syntax

DTWKDATX_{w.}

Arguments

w
specifies the width of the output field.

Default 29

Range 3–37

Details

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

day-of-week

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

dd

is an integer that represents the day of the month.

month-name

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

yy or *yyyy*

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

Comparisons

The DTWKDATX_{w.} format is similar to the WEEKDATX_{w.} format in that they both write date values. The difference is that DTWKDATX_{w.} expects a SAS datetime value as input, and WEEKDATX_{w.} expects a SAS date value.

Example

The example table uses as input the value 1678898894, which is the SAS datetime value that corresponds to March 14, 2013, at 04:48:14 PM.

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

See Also

Formats:

- [“DATEw. Format” on page 98](#)
- [“WEEKDATEw. Format” on page 154](#)
- [“WEEKDATXw. Format” on page 155](#)

DTYEARw. Format

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

Category: Date and Time

Alignment: Right

Syntax

DTYEARw.

Arguments

w specifies the width of the output field.

Default 4

Range 2–4

Details

remove after conversion

Comparisons

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

Example

The example table uses as input the value 1678898894, which is the SAS datetime value that corresponds to March 14, 2013, at 04:48:14 AM.

Statements	Results
	-----1
<code>select put(1678898894,dtyear.);</code>	2013
<code>select put(1678898894,dtyear2.);</code>	13
<code>select put(1678898894,dtyear3.);</code>	13
<code>select put(1678898894,dtyear4.);</code>	2013

See Also

Formats:

- [“DATEw. Format” on page 98](#)
- [“DATETIMEw.d Format” on page 101](#)
- [“YEARw. Format” on page 158](#)

DTYYQCw. Format

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

Category: Date and Time

Alignment: Right

Syntax

DTYYQC^w.

Arguments

^w
specifies the width of the output field.

Default 4

Range 4–6

Details

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

Example

The example table uses as input the value 1678898894, which is the SAS datetime value that corresponds to March 14, 2013, at 04:48:52 PM.

Statements	Results
	-----1
select put(1678898894,dtqqc.);	13:1
select put(1678898894,dtqqc4.);	13:1
select put(1678898894,dtqqc5.);	13:1
select put(1678898894,dtqqc6.);	2013:1

See Also

Formats:

- [“DATEw. Format” on page 98](#)
- [“DATETIMEw.d Format” on page 101](#)

Ew. Format

Writes numeric values in scientific notation.

Category: Numeric

Alignment: Right

Syntax

Ew.

Arguments

w
specifies the width of the output field.

Default 12

Range 7–32

Details

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

Example

Statements	Results
	-----1-----
<code>select put(1257,e10.);</code>	1.257E+03
<code>select put(-1257,e10.);</code>	-1.257E+03

EUROW.d Format

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

Category: Numeric

Alignment: Right

Syntax

EUROW.d

Arguments

w

specifies the width of the output field.

Default 6

Range 1–32

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

d

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

Default 0

Range 0–31

Requirement must be less than w

Comparisons

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

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

Example

These examples use 1254.71 as the value of amount.

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

See Also

Formats:

- [“DOLLARw.d Format” on page 107](#)
- [“EUROXw.d Format” on page 118](#)
- [“YENw.d Format” on page 159](#)

EUROXw.d Format

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

Category:	Numeric
Alignment:	Right

Syntax

EUROXw.d

Arguments

w
specifies the width of the output field.

Default 6

Range 1–32

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

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

Default	0
Range	0–31
Requirement	must be less than <i>w</i>

Comparisons

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

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

Example

These examples use 1254.71 as the value of amount.

Statements	Results
	-----1-----2-----3
<code>select put(1254.71,eurox10.2);</code>	E1.254,71
<code>select put(1254.71,eurox5.);</code>	1.255
<code>select put(1254.71,eurox9.2);</code>	E1.254,71
<code>select put(1254.71,eurox15.3);</code>	E1.254,710

See Also

Formats:

- [“EUROw.d Format” on page 117](#)

FLOATw.d Format

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

Category: Numeric

Alignment: Left

Syntax

FLOAT w . d

Arguments

w
specifies the width of the output field.

Requirement width must be 4

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

Default 0

Range 0–31

Details

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

Example

Statements	Results*
<pre>select put(1,float4.);</pre>	0000803F

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

See Also

Formats:

- [“IEEE \$w\$. \$d\$ Format” on page 125](#)

FRACT w . Format

Converts numeric values to fractions.

Category: Numeric

Alignment: Right

Syntax

FRACT_w.

Arguments

w
specifies the width of the output field.

Default 10

Range 4–32

Details

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

Example

Statements	Results
	-----1
select put(0.6666666667,fract8.);	2/3
select put(0.2784,fract8.);	174/625

HEXw. Format

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

Category: Numeric

Alignment: Left

Syntax

HEX_w.

Arguments

w
specifies the width of the output field.

Default 8

Range 1–16

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

Details

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

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

Comparisons

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

Example

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

See Also

- Formats:**
- [“BINARY \$w\$. Format” on page 93](#)
 - [“\\$HEX \$w\$. Format” on page 83](#)

HHMM w . d Format

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

Category: Date and Time

Alignment: Right

Syntax

HHMMw.d

Arguments

w specifies the width of the output field.

Default	5
Range	2–20

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

Default	0
Range	0–19
Requirement	must be less than w

Details

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

hh is an integer.

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

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

Comparisons

The HHMMw.d format is similar to the TIMEw.d format except that the HHMMw.d format does not print seconds.

The HHMMw.d format and the TIMEw.d format write a leading blank for the single-hour digit. The TODw.d format writes a leading zero for a single-hour digit.

Example

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

Statements	Results
	----+----1
select put(46796,hhmm.);	13:00
select put(46796,hhmm8.2);	12:59.93

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

See Also

Formats:

- “[HOURw.d Format](#)” on page 124
- “[MMSSw.d Format](#)” on page 130
- “[TIMEw.d Format](#)” on page 148
- “[TODw.d Format](#)” on page 151

Functions:

- “[HOUR Function](#)” on page 245
- “[MINUTE Function](#)” on page 272
- “[SECOND Function](#)” on page 295

HOURw.d Format

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

Category: Date and Time

Alignment: Right

Syntax

HOURw.*d*

Arguments

w
specifies the width of the output field.

Default 2

Range 2–20

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

Range 0–19

Requirement must be less than *w*

Details

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

Example

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

Statements	Results
	-----+-----1
<pre>select put(41400, hour4.1);</pre>	11.5

See Also

Formats:

- [“HHMMw.d Format” on page 122](#)
- [“MMSSw.d Format” on page 130](#)
- [“TIMEw.d Format” on page 148](#)
- [“TODw.d Format” on page 151](#)

IEEEw.d Format

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

Category: Numeric

Alignment: Left

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

Syntax

IEEEw.*d*

Arguments

w
specifies the width of the output field.

Default 8

Range 3–8

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

d
specifies to multiply the number by 10^{*d*}.

Default 0
Range 0–10

Details

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

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

Example

Statements	Results*
<code>select put(1,ieee4.);</code>	3FF00000
<code>select put(1,ieee5.);</code>	3FF0000000

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

See Also

Formats:

- [“FLOAT*w.d* Format” on page 119](#)

JULIAN*w*. Format

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

Category: Date and Time
Alignment: Left

Syntax

JULIAN*w*.

Arguments

w
specifies the width of the output field.

Default 5

Range 5–7

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

Details

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

yy or *yyyy*

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

ddd

is the number of the day, 1–365 (or 1–366 for leap years), in that year.

Example

The example table uses the input value of 19431, which is the SAS date value that corresponds to March 14, 2013 (the 73rd day of the year).

Statements	Results
	-----1
<pre>select put(19431,julian5.);</pre>	13073
<pre>select put(19431,julian7.);</pre>	2013073

See Also

Formats:

- [“DATEw. Format” on page 98](#)

MMDDYYw. Format

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

Category: Date and Time

Alignment: Right

Syntax

MMDDYYw.

Arguments

w	specifies the width of the output field.
Default	8
Range	2–10
Interaction	When <i>w</i> has a value of from 2 to 5, the date appears with as much of the month and the day as possible. When <i>w</i> is 7, the date appears as a two-digit year without slashes.

Details

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

- mm*
is an integer that represents the month.
- /
is the separator.
- dd*
is an integer that represents the day of the month.
- [*yy*]*yy*
is a two-digit or four-digit integer that represents the year.

Example

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

Statements	Results
	-----1-----
<code>select put(19431,mmddy2.);</code>	03
<code>select put(19431,mmddy3.);</code>	03
<code>select put(19431,mmddy4.);</code>	0314
<code>select put(19431,mmddy5.);</code>	03/14
<code>select put(19431,mmddy6.);</code>	031413
<code>select put(19431,mmddy7.);</code>	031413
<code>select put(19431,mmddy8.);</code>	03/14/13
<code>select put(19431,mmddy10.);</code>	03/14/2013

See Also

Formats:

- [“DATEw. Format” on page 98](#)
- [“DDMMYYw. Format” on page 104](#)
- [“MMDDYYxw. Format” on page 129](#)
- [“YYMMDDw. Format” on page 163](#)

MMDDYYxw. Format

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

Category:	Date and Time
Alignment:	Right

Syntax

MMDDYYxw.

Arguments

x	identifies a separator or specifies that no separator appear between the month, the day, and the year. Valid values for x are any of the following:
B	separates with a blank
C	separates with a colon
D	separates with a hyphen
N	indicates no separator
P	separates with a period
S	separates with a slash.
w	specifies the width of the output field.
Default	8
Range	2–10
Interactions	When w has a value of from 2 to 5, the date appears with as much of the month and the day as possible. When w is 7, the date appears as a two-digit year without separators.

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

Details

The MMDDYY xw . format writes SAS date values in the form *mmd**d*[*yy*]*yy* or *mmXddX*[*yy*]*yy*, where

- mm*
is an integer that represents the month.
- X*
is a specified separator.
- dd*
is an integer that represents the day of the month.
- [*yy*]*yy*
is a two-digit or four-digit integer that represents the year.

Example

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

Statements	Results
	----+----1----
select put(19431,mmddyyc5.);	03:14
select put(19431,mmddyid8.);	03-14-13
select put(19431,mmddyyn8.);	03132013
select put(19431,mmddyyp10.);	03.14.2013

See Also

Formats:

- [“DATEw. Format” on page 98](#)
- [“DDMMYYxw. Format” on page 105](#)
- [“MMDDYYw. Format” on page 127](#)
- [“YYMMDDxw. Format” on page 164](#)

MMSSw.d Format

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

Category: Date and Time

Alignment: Right

Syntax

MMSS w . d

Arguments

w	specifies the width of the output field.
Default	5
Range	2–20
Tip	Set w to a minimum of 5 to write a value that represents minutes and seconds.
d	specifies the number of digits to the right of the decimal point in the seconds value. Therefore, the SAS time value includes fractional seconds.
Range	0–19
Restriction	must be less than w

Example

The example table uses the SAS input value of 4530.

Statements	Results
	-----1
<pre>select put(4530,mmss.);</pre>	75:30

See Also

Formats:

- [“HHMMw.d Format” on page 122](#)
- [“TIMEw.d Format” on page 148](#)

MMYYw. Format

Writes SAS date values in the form *mmM[yy]yy*, where M is the separator and the year appears as either 2 or 4 digits.

Category: Date and Time

Alignment: Right

Syntax

MMYY^w.

Arguments

w	specifies the width of the output field.
Default	7
Range	5–32
Interaction	When <i>w</i> has a value of 5 or 6, the date appears with only the last two digits of the year. When <i>w</i> is 7 or more, the date appears with a four-digit year.

Details

The MMYY^w. format writes SAS date values in the form *mm*M<*yy*>*yy*, where

mm
is an integer that represents the month.

M
is the character separator.

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

Example

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

Statements	Results
	-----1-----
select put(19431,mmyy5.);	03M13
select put(19431,mmyy6.);	03M13
select put(19431,mmyy7.);	03M2013
select put(19431,mmyy10.);	03M2013

See Also

Formats:

- “DATE^w. Format” on page 98
- “MMYY^{xw}. Format” on page 133
- “YYMM^w. Format” on page 160

MMYYxw. Format

Writes SAS date values in the form *mm[yy]yy* or *mmX[yy]yy*. The *x* in the format name is a character that represents the special character. The special character separates the month and the year. That special character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits.

Category: Date and Time

Alignment: Right

Syntax

MMYY^{xw}.

Arguments

- x**
- identifies a separator or specifies that no separator appear between the month and the year. Valid values for *x* are any of the following:
 - C
 - separates with a colon
 - D
 - separates with a hyphen
 - N
 - indicates no separator
 - P
 - separates with a period
 - S
 - separates with a forward slash.

w

- specifies the width of the output field.

Default	7
Range	5–32
Interactions	<p>When <i>x</i> is set to N, no separator is specified. The width range is then 4–32, and the default changes to 6.</p> <p>When <i>x</i> has a value of C, D, P, or S and <i>w</i> has a value of 5 or 6, the date appears with only the last two digits of the year. When <i>w</i> is 7 or more, the date appears with a four-digit year.</p> <p>When <i>x</i> has a value of N and <i>w</i> has a value of 4 or 5, the date appears with only the last two digits of the year. When <i>x</i> has a value of N and <i>w</i> is 6 or more, the date appears with a four-digit year.</p>

Details

The MMYY w . format writes SAS date values in the form $mm[yy]yy$ or $mmX[yy]yy$, where

- mm is an integer that represents the month.
- X is a specified separator.
- $[yy]yy$ is a two-digit or four-digit integer that represents the year.

Example

The following examples use the input value of 19560, which is the SAS date value that corresponds to July 14, 2013.

Statements	Results
	----+----1----
<code>select put(19560,mmmyc5.);</code>	07:13
<code>select put(19560,mmyyd7.);</code>	07-2013
<code>select put(19560,mmyyn4.);</code>	0713
<code>select put(19560,mmyyp8.);</code>	07.2013
<code>select put(19560,mmyys10.);</code>	07/2013

See Also

Formats:

- [“DATEw. Format” on page 98](#)
- [“MMYYw. Format” on page 131](#)
- [“YYMMw. Format” on page 160](#)

MONNAMEw. Format

Writes SAS date values as the name of the month.

Category: Date and Time

Alignment: Right

Syntax

MONNAME w .

Arguments

w

specifies the width of the output field.

Default 9

Range 1–32

Tip Use MONNAME3. to print the first three letters of the month name.

Details

If necessary, SAS truncates the name of the month to fit the format width.

Example

The example table uses the input value of 19431, which is the SAS date value that corresponds to March 14, 2014.

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

See Also

Formats:

- [“MONTHw. Format” on page 135](#)

MONTHw. Format

Writes SAS date values as the month of the year.

Category: Date and Time

Alignment: Right

Syntax

MONTHw.

Arguments

w

specifies the width of the output field.

Default	2
Range	1–32

Details

The MONTH_w. format writes the month (1 through 12) of the year from a SAS date value.

Example

The example table uses the input value of 19431, which is the SAS date value that corresponds to March 14, 2014.

Statements	Results
	-----1
<pre>select put(19431,month.);</pre>	3

See Also

- Formats:
- [“MONNAME_w. Format” on page 134](#)

MONYY_w. Format

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

Category:	Date and Time
Alignment:	Right

Syntax

MONYY_w.

Arguments

w
specifies the width of the output field.

Default	5
Range	5–7

Details

The MONYY_w. format writes SAS date values in the form *mmmyy* or *mmmyyyy*, where

mmm

is the first three letters of the month name.

yy or *yyyy*

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

Comparisons

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

Example

The example table uses the input value of 19704, which is the SAS date value that corresponds to December 12, 2013.

Statements	Results
	-----1
<code>select put(19704,monyy5.);</code>	DEC13
<code>select put(19704,monyy7.);</code>	DEC2013

See Also

Formats:

- [“DATEw. Format” on page 98](#)
- [“DTMONYYw. Format” on page 111](#)
- [“DDMMYYw. Format” on page 104](#)
- [“MMDDYYw. Format” on page 127](#)
- [“YYMMDDw. Format” on page 163](#)

Functions:

- [“MONTH Function” on page 275](#)
- [“YEAR Function” on page 318](#)

NEGPARENw.d Format

Writes negative numeric values in parentheses.

Category: Numeric

Alignment: Right

Syntax

NEGPAREN w . d

Arguments

- w
specifies the width of the output field.

Default 6

Range 1–32
- d
specifies the number of digits to the right of the decimal point in the numeric value.

Default 0

Range 0–31

Details

The NEGPAREN w . d format attempts to right align output values. If the input value is negative, NEGPAREN w . d displays the output by enclosing the value in parentheses, if the field that you specify is wide enough. Otherwise, it uses a minus sign to represent the negative value. If the input value is non-negative, NEGPAREN w . d displays the value with a leading and trailing blank to ensure proper column alignment. It reserves the last column for a close parenthesis even when the value is positive.

Comparisons

The NEGPAREN w . d format is similar to the COMMA w . d format in that it separates every three digits of the value with a comma.

Example

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

NENGOW. Format

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

Category: Date and Time
Alignment: Left

Syntax

NENGOW.

Arguments

w
 specifies the width of the output field.

Default 10

Range 2–10

Details

The NENGOW. format writes SAS date values in the form *e.yymmdd*, where

e
 is the first letter of the name of the emperor (Meiji, Taisho, Showa, or Heisei).

yy
 is an integer that represents the year.

mm
 is an integer that represents the month.

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

If the width is too small, SAS omits the period.

Example

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

Statements	Results
	-----1
select put(19431,nengo3.);	H25
select put(19431,nengo6.);	H25/03
select put(19431,nengo8.);	H.250314
select put(19431,nengo9.);	H25/03/14
select put(19431,nengo10.);	H.25/03/14

See Also

Formats:

- [“DATEw. Format” on page 98](#)

OCTALw. Format

Converts numeric values to octal representation.

Category: Numeric

Alignment: Left

Syntax

OCTALw.

Arguments

w
specifies the width of the output field.

Default 3

Range 1–24

Details

If necessary, the OCTALw. format converts numeric values to integers before displaying them in octal representation.

Comparisons

OCTALw. converts numeric values to octal representation. The \$OCTALw. format converts character values to octal representation.

Example

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

See Also

Formats:

- [“\\$OCTALw. Format” on page 84](#)

PERCENTw.d Format

Writes numeric values as percentages.

Category: Numeric

Alignment: Right

Syntax

PERCENTw.*d*

Arguments

w
specifies the width of the output field.

Default 6

Range 4–32

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

Range 0–31

Requirement must be less than *w*

Details

The PERCENTw.d format multiplies values by 100, formats them the same as the BESTw.d format, and adds a percent sign (%) to the end of the formatted value, while it encloses negative values in parentheses. The PERCENTw.d format allows room for a percent sign and parentheses, even if the value is not negative.

Example

Statements	Results
	-----1-----2
<code>select put(0.1,percent10.);</code>	10%
<code>select put(1.2,percent10.);</code>	120%
<code>select put(-.05,percent10.);</code>	(5%)

See Also

Formats:

- “PERCENTNw.d Format” on page 142

PERCENTNw.d Format

Produces percentages, using a minus sign for negative values.

Category: Numeric

Alignment: Right

Syntax

PERCENTNw.*d*

Arguments

w
specifies the width of the output field.

Default 6

Range 4–32

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

Range 0–31

Requirement must be less than *w*

Details

The PERCENTNw.d format multiplies negative values by 100, formats them the same as the BESTw.d format, adds a minus sign to the beginning of the value, and adds a percent sign (%) to the end of the formatted value. The PERCENTNw.d format allows room for a percent sign and a minus sign, even if the value is not negative.

Comparisons

The PERCENTNw.d format produces percents by using a minus sign instead of parentheses for negative values. The PERCENTw.d format produces percents by using parentheses for negative values.

Example

Statements	Results
<code>select put(-0.1,percentn10.);</code>	-10%
<code>select put(.2,percentn10.);</code>	20%
<code>select put(.8,percentn10.);</code>	80%

Statements	Results
<code>select put(-0.05,percentn10.);</code>	-5%
<code>select put(-6.3,percentn10.);</code>	-630%

See Also

Formats:

- [“PERCENTw.d Format” on page 141](#)

QTRw. Format

Writes SAS date values as the quarter of the year.

Category: Date and Time

Alignment: Right

Syntax

QTR^w.

Arguments

w
specifies the width of the output field.

Default 1

Range 1–32

Example

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

Statements	Results
	-----1
<code>select put(19431,qtr.);</code>	1

See Also

Formats:

- [“DATEw. Format” on page 98](#)

- “QTRRw. Format” on page 144
- “YYQw. Format” on page 167
- “YYQxw. Format” on page 168
- “YYQZw. Format” on page 173

QTRRw. Format

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

Category: Date and Time

Alignment: Right

Syntax

QTRR_w.

Arguments

w
specifies the width of the output field.

Default 3
Range 3–32

Example

The example table uses the input value of 19624, which is the SAS date value that corresponds to September 23, 2013.

Statements	Results
	-----1
select put(19624,qtrr.);	III

See Also

Formats:

- “QTRw. Format” on page 143
- “YYQRw. Format” on page 170
- “YYQRxw. Format” on page 171

ROMANw. Format

Writes numeric values as roman numerals.

Category: Numeric

Alignment: Left

Syntax

ROMANw.

Arguments

w
specifies the width of the output field.

Default 6

Range 2–32

Details

The ROMANw. format truncates a floating-point value to its integer component before the value is written.

Example

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

See Also

Formats:

- “DATEw. Format” on page 98
- “JULIANw. Format” on page 126

SIZEKw.d Format

Writes a numeric value in the form nK for kilobytes.

Category: Numeric

Alignment: Right

Syntax

SIZEK w . d

Arguments

w
specifies the width of the output field.

Default 9
Range 2–33

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

Default 0
Range 0–31

Details

To write a numeric value in the form nK by using the SIZEK $w.d$ format, the value of n is calculated by dividing the numeric value by 1,024. The symbol K indicates that the value is a multiple of 1,024.

Example

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

See Also

Formats:

- [“SIZEKMGw.d Format” on page 146](#)

SIZEKMGw.d Format

Writes a numeric value in the form nKB for kilobytes, nMB for megabytes, or nGB for gigabytes.

Category: Numeric

Alignment: Right

Syntax

SIZEKMGw.*d*

Arguments

w
specifies the width of the output field.

Default	9
Range	2–33

d
specifies the number of digits to the right of the decimal point in the numeric value. This argument is optional.

Default	0
Range	0–31

Details

When you specify the SIZEKMGw.d format, SAS determines the best suffix: KB for kilobytes; MB for megabytes; or GB for gigabytes; and divides the SAS numeric value by one of the following values:

KB	1024
MB	1048576
GB	1073741824

Example

Statements	Results
	-----1
select put (3688, sizekmg.);	4KB
select put (1048576, sizekmg.);	1MB
select put (83409922345, sizekmg.);	8GB

See Also

Formats:

- [“SIZEKw.d Format” on page 145](#)

TIMEw.d Format

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

Category: Date and Time

Alignment: Right

Syntax

TIMEw.*d*

Arguments

<i>w</i>	specifies the width of the output field.
Default	8
Range	2–20
Tip	Make <i>w</i> large enough to produce the desired results. To obtain a complete time value with three decimal places, you must allow at least 12 spaces: eight spaces to the left of the decimal point, one space for the decimal point itself, and three spaces for the decimal fraction of seconds.

<i>d</i>	specifies the number of digits to the right of the decimal point in the seconds value.
Default	0
Range	0–19
Requirement	must be less than <i>w</i>

Details

The TIMEw.d format writes SAS time values in the form *hh:mm:ss.ss*, where

hh
is an integer.

Note: If *hh* is a single digit, TIMEw.d places a leading blank before the digit. For example, the TIMEw.d. format writes 9:00 instead of 09:00.

mm
is the number of minutes, ranging from 00 through 59.

ss.ss
is the number of seconds, ranging from 00 through 59, with the fraction of a second following the decimal point.

Comparisons

The TIMEw.d format is similar to the HHMMw.d format except that TIMEw.d includes seconds.

The TIMEw.d format and the HHMMw format write a leading blank for a single-hour digit. The TODw.d format writes a leading zero for a single-hour digit.

Example

The example table uses the input value of 59083, which is the SAS time value that corresponds to 4:24:43 PM.

Statements	Results
	-----1
select put(59083,time.);	16:24:43

See Also

Formats:

- [“HHMMw.d Format” on page 122](#)
- [“HOURw.d Format” on page 124](#)
- [“MMSSw.d Format” on page 130](#)
- [“TODw.d Format” on page 151](#)

Functions:

- [“HOUR Function” on page 245](#)
- [“MINUTE Function” on page 272](#)
- [“SECOND Function” on page 295](#)

TIMEAMPMw.d Format

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

Category: Date and Time

Alignment: Right

Syntax

TIMEAMPMw.[d]

Arguments

w
specifies the width of the output field.

Default	11
Range	2–20

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

Default	0
Range	0–19
Requirement	must be less than <i>w</i>

Details

The TIMEAMPM $w.d$ format writes SAS time values in the form *hh:mm:ss.ss* with AM or PM, where

hh
is an integer that represents the hour.

mm
is an integer that represents the minutes.

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

Times greater than 23:59:59 PM appear as the next day.

Make *w* large enough to produce the desired results. To obtain a complete time value with three decimal places and AM or PM, you must allow at least 11 spaces (*hh:mm:ss* PM). If *w* is less than 5, SAS writes AM or PM only.

Comparisons

- The TIMEAMPMM $w.d$ format is similar to the TIMEM $w.d$ format except, that TIMEAMPMM $w.d$ prints AM or PM at the end of the time.
- TIME $w.d$ writes hours greater than 11:59:59, and TIMEAMPM $w.d$ does not.

Example

The example table uses the input value of 59083, which is the SAS time value that corresponds to 4:24:43 PM.

Statements	Results
	-----1-----+
select put(59083,timeampm3.);	PM
select put(59083,timeampm5.);	4 PM
select put(59083,timeampm7.);	4:24 PM
select put(59083,timeampm11.);	4:24:43 PM

See Also

Formats:

- [“DATEAMPMw.d Format” on page 99](#)
- [“TIMEw.d Format” on page 148](#)

TODw.d Format

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

Category: Date and Time

Alignment: Right

Syntax

TODw.[\[d\]](#)

Arguments

w

specifies the width of the output field.

Default 8

Range 2–20

Tip SAS writes a zero for a zero hour if the specified width is sufficient, for example, 02:30 or 00:30.

d

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

Default 0

Range 0–19

Requirement must be less than *w*

Details

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

hh

is an integer that represents the hour.

mm

is an integer that represents the minutes.

ss.ss

is the number of seconds to two decimal places.

Comparisons

The TOWw.d format writes a leading zero for a single-hour digit. The TIMEw.d format and the HHMMw.d format write a leading blank for a single-hour digit.

Example

Statements	Results
<pre>select put(1472049623,tod9.);</pre>	14:40:23

See Also

Formats:

- [“TIMEw.d Format” on page 148](#)
- [“TIMEAMP Mw.d Format” on page 149](#)

VAXRBw.d Format

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

Category: Numeric

Alignment: Right

Syntax

VAXRB_w._d

Arguments

w
specifies the width of the output field.

Default 8

Range 2–8

d
specifies the power of 10 by which to divide the value.

Default 0

Range 0–31

Details

Use the VAXRB_w._d format to write data in native VAX/VMS floating-point notation.

Example

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

* The result is the hexadecimal representation for the integer.

w.d Format

Writes standard numeric data one digit per byte.

Category: Numeric

Alignment: Right

Alias: Fw.d

Syntax

w.[*d*]

Arguments

w

specifies the width of the output field.

Range 1–32

Tip

Allow enough space to write the value, the decimal point, and a minus sign, if necessary.

d

specifies the number of digits to the right of the decimal point in the numeric value. This argument is optional.

Range 0–31

Requirement must be less than *w*

Tip

If *d* is 0 or you omit *d*, *w.d* writes the value without a decimal point.

Details

The *w.d* format rounds to the nearest number that fits in the output field. If *w.d* is too small, SAS might shift the decimal to the BEST*w*. format. The *w.d* format writes negative numbers with leading minus signs. In addition, *w.d* right aligns before writing and pads the output with leading blanks.

Comparisons

The *Zw.d* format is similar to the *w.d* format except that *Zw.d* pads right-aligned output with 0s instead of blanks.

Example

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

See Also

Formats:

- [“BESTw. Format” on page 90](#)
- [“BESTDw.p Format” on page 92](#)
- [“Dw.p Format” on page 96](#)
- [“\\$w. Format” on page 89](#)
- [“Zw.d Format” on page 174](#)

WEEKDATEw. Format

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

Category: Date and Time

Alignment: Right

Syntax

WEEKDATE^{w.}

Arguments

w
specifies the width of the output field.

Default 29

Range 3–37

Details

The WEEKDATE^{w.} format writes SAS date values in the form *day-of-week, month-name dd, yy* (or *yyyy*), where

dd

is an integer that represents the day of the month.

yy or *yyyy*

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

If *w* is too small to write the complete day of the week and month, SAS abbreviates as needed.

Comparisons

The WEEKDATE*w*. format is the same as the WEEKDATX*w*. format except that WEEKDATX*w*. prints *dd* before the month's name.

Example

The example table uses the input value of 19537 which is the SAS date value that corresponds to June 28, 2013.

Statements	Results
	-----1-----2
<code>select put(19537,weekdate3.);</code>	Fri
<code>select put(19537,weekdate9.);</code>	Friday
<code>select put(19537,weekdate15.);</code>	Fri, Jun 28, 13
<code>select put(19537,weekdate17.);</code>	Fri, Jun 28, 2013

See Also

Formats:

- [“DTWKDATXw. Format” on page 113](#)
- [“DATEw. Format” on page 98](#)
- [“DDMMYYw. Format” on page 104](#)
- [“MMDDYYw. Format” on page 127](#)
- [“TODw.d Format” on page 151](#)
- [“WEEKDATXw. Format” on page 155](#)
- [“YYMMDDw. Format” on page 163](#)

WEEKDATXw. Format

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

Category: Date and Time

Alignment: Right

Syntax

WEEKDATX^w.

Arguments

^w
specifies the width of the output field.

Default 29

Range 3–37

Details

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

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

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

If *w* is too small to write the complete day of the week and month, then SAS abbreviates as needed.

Comparisons

The WEEKDATE^w. format is the same as the WEEKDATX^w. format, except that WEEKDATE^w. prints *dd* after the month's name.

The WEEKDATX^w. format is the same as the DTWKDATX^w. format, except that DTWKDATX^w. expects a datetime value as input.

Example

The example table uses the input value of 19405, which is the SAS date value that corresponds to February 16, 2013.

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

See Also

Formats:

- “DTWKDATX^w. Format” on page 113
- “DATE^w. Format” on page 98
- “DDMMYY^w. Format” on page 104

- [“MMDDYYw. Format” on page 127](#)
- [“TODw.d Format” on page 151](#)
- [“WEEKDATEw. Format” on page 154](#)
- [“YYMMDDw. Format” on page 163](#)

WEEKDAYw. Format

Writes SAS date values as the day of the week.

Category: Date and Time

Alignment: Right

Syntax

WEEKDAYw.

Arguments

w
specifies the width of the output field.

Default 1

Range 1–32

Details

The WEEKDAYw. format writes a SAS date value as the day of the week (where 1=Sunday, 2=Monday, and so on).

Example

The example table uses the input value of 19405, which is the SAS date value that corresponds to February 16, 2013.

Statements	Results
	-----1
select put(19405,weekday.);	7

See Also

Formats:

- [“DOWNAMEw. Format” on page 109](#)

YEARw. Format

Writes SAS date values as the year.

Category: Date and Time

Alignment: Right

Syntax

YEAR_w.

Arguments

w
specifies the width of the output field.

Default 4

Range 2–32

Tip If *w* is less than 4, the last two digits of the year print. Otherwise, the year value prints as four digits.

Comparisons

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

Example

The example table uses the input value of 19537, which is the SAS date value that corresponds to June 28, 2013.

Statements	Results
	----+----1
select put(19537,year2.);	13
select put(19537,year4.);	2013

See Also

Formats:

- [“DTYEARw. Format” on page 114](#)

YENw.d Format

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

Category: Numeric

Alignment: Right

Syntax

YENw.d

Arguments

w
specifies the width of the output field.

Default 1

Range 1–32

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

Restriction must be either 0 or 2

Tip If *d* is 2, then YENw.d writes a decimal point and two decimal digits. If *d* is 0, then YENw.d does not write a decimal point or decimal digits.

Details

The YENw.d format writes numeric values with a leading yen sign and with a comma that separates every three digits of each value.

The hexadecimal representation of the code for the yen sign character is 5B on EBCDIC systems and 5C on ASCII systems. The monetary character these codes represent can be different in other countries.

Example

Statements	Results
	-----1
<code>select put(1254.71,yen10.2);</code>	¥1,254.71

See Also

Formats:

- [“DOLLARw.d Format” on page 107](#)

- “EUROw.d Format” on page 117

YYMMw. Format

Writes SAS date values in the form [yy]yyMmm, where M is the separator and the year appears as either 2 or 4 digits.

Category: Date and Time

Alignment: Right

Syntax

YYMMw.

Arguments

w	specifies the width of the output field.
Default	7
Range	5–32
Interaction	When <i>w</i> has a value of 5 or 6, the date appears with only the last two digits of the year. When <i>w</i> is 7 or more, the date appears with a four-digit year.

Details

The YYMMw. format writes SAS date values in the form [yy]yyMmm, where

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

M
is the character separator.

mm
is an integer that represents the month.

Example

The following examples use the input value of 19656, which is the SAS date value that corresponds to October 25, 2013.

Statements	Results
	----+----1----
select put(19656, yymm5.);	13M10
select put(19656, yymm6.);	13M10

Statements	Results
<pre>select put(19656,yymm.);</pre>	2013M10
<pre>select put(19656,yymm7.);</pre>	2013M10
<pre>select put(19656,yymm10.);</pre>	2013M10

See Also

Formats:

- [“DATEw. Format” on page 98](#)
- [“MMYYw. Format” on page 131](#)
- [“YYMMxw. Format” on page 161](#)

YYMMxw. Format

Writes SAS date values in the form [yy]yy mm or [yy]yy- mm . The x in the format name represents the special character that separates the year and the month. This special character can be a hyphen (-), period (.), slash(/), colon(:), or no separator. The year can be either 2 or 4 digits.

Category: Date and Time

Alignment: Right

Syntax

YYMM xw .

Arguments

- x
- identifies a separator or specifies that no separator appear between the year and the month. Valid values for x are any of the following:
- C
separates with a colon
 - D
separates with a hyphen
 - N
indicates no separator
 - P
separates with a period
 - S
separates with a forward slash
- w
- specifies the width of the output field.

Default	7
Range	5–32
Interactions	<p>When <i>x</i> is set to <i>N</i>, no separator is specified. The width range is then 4–32, and the default changes to 6.</p> <p>When <i>x</i> has a value of C, D, P, or S and <i>w</i> has a value of 5 or 6, the date appears with only the last two digits of the year. When <i>w</i> is 7 or more, the date appears with a four-digit year.</p> <p>When <i>x</i> has a value of <i>N</i> and <i>w</i> has a value of 4 or 5, the date appears with only the last two digits of the year. When <i>x</i> has a value of <i>N</i> and <i>w</i> is 6 or more, the date appears with a four-digit year.</p>

Details

The YYMMXw. format writes SAS date values in one of the following forms:

- yymmdd*
[*yy*]*yy-mmxxdd*
<*yy*>*yy*
is a two-digit or four-digit integer that represents the year.
- x*
is a specified separator.
- mm*
is an integer that represents the month.

Example

The following examples use the input value of 19537, which is the SAS date value that corresponds to June 28, 2013.

Statements	Results
	----+----1----
select put(19537, yymmc5.);	13:06
select put(19537, yymmd.);	2013-06
select put(19537, yymmn4.);	1306
select put(19537, yymmp8.);	2013.06
select put(19537, yymms10.);	2013/06

See Also

Formats:

- [“DATEw. Format” on page 98](#)

- “MMYYxw. Format” on page 133
- “YYMMw. Format” on page 160

YYMMDDw. Format

Writes SAS date values in the form *yymmdd* or *[yy]yy-mm-dd*, where a hyphen (-) is the separator and the year appears as either 2 or 4 digits.

Category: Date and Time

Alignment: Right

Syntax

YYMMDDw.

Arguments

w

specifies the width of the output field.

Default	8
----------------	---

Range	2–10
--------------	------

Interaction	When <i>w</i> has a value of from 2 to 5, the date appears with as much of the year and the month as possible. When <i>w</i> is 7, the date appears as a two-digit year without a hyphen.
--------------------	---

Details

The YYMMDDw. format writes SAS date values in one of the following forms:

yymmdd

[yy]yy-mm-dd

[yy]yy

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

-

is the separator.

mm

is an integer that represents the month.

dd

is an integer that represents the day of the month.

To format a date that has a four-digit year and no separators, use the YYMMDDx. format.

Example

The following examples use the input value of 19450, which is the SAS date value that corresponds to April 2, 2013.

Statements	Results
	-----1-----
<code>select put(19450,yymmdd2.);</code>	13
<code>select put(19450,yymmdd3.);</code>	13
<code>select put(19450,yymmdd4.);</code>	1304
<code>select put(19450,yymmdd5.);</code>	13-04
<code>select put(19450,yymmdd6.);</code>	130402
<code>select put(19450,yymmdd7.);</code>	130402
<code>select put(19450,yymmdd8.);</code>	13-04-02
<code>select put(19450,yymmdd10.);</code>	2013-04-02

See Also

Formats:

- [“DATEw. Format” on page 98](#)
- [“DDMMYYw. Format” on page 104](#)
- [“MMDDYYw. Format” on page 127](#)
- [“YYMMDDxw. Format” on page 164](#)

Functions:

- [“DAY Function” on page 231](#)
- [“MONTH Function” on page 275](#)
- [“YEAR Function” on page 318](#)

YYMMDDxw. Format

Writes date values in the form `[yy]yymmdd` or `[yy]yy-mm-dd`. The *x* in the format name is a character that represents the special character which separates the year, month, and day. This special character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits.

Category: Date and Time

Alignment: Right

Syntax

`YYMMDD`[xw.](#)

Arguments

x

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

B

separates with a blank

C

separates with a colon

D

separates with a hyphen

N

indicates no separator

P

separates with a period

S

separates with a slash.

w

specifies the width of the output field.

Default	8
----------------	---

Range	2–10
--------------	------

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

	When <i>x</i> has a value of N, the width range is 2–8.
--	---

Details

The YYMMDDxw. format writes SAS date values in one of the following forms:

yymmdd

[*yy*]*yxmmd*

where

[*yy*]*yy*

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

x

is a specified separator.

mm

is an integer that represents the month.

dd

is an integer that represents the day of the month.

Example

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

Statements	Results
	-----1-----
<code>select put(19704, yymmddc5.);</code>	13:12
<code>select put(19704, yymmddd8.);</code>	13-12-12
<code>select put(19704, yymmddn8.);</code>	20131212
<code>select put(19704, yymmddp10.);</code>	2013.12.12

See Also

Formats:

- “DATEw. Format” on page 98
- “DDMMYYxw. Format” on page 105
- “MMDDYYxw. Format” on page 129
- “YYMMDDw. Format” on page 163

Functions:

- “DAY Function” on page 231
- “MONTH Function” on page 275
- “YEAR Function” on page 318

YYMONw. Format

Writes SAS date values in the form *yymm* or *yyyymm*.

Category: Date and Time

Alignment: Right

Syntax

YYMON_w.

Arguments

w

specifies the width of the output field. If the format width is too small to print a four-digit year, only the last two digits of the year are printed.

Default 7

Range 5–32

Details

The YYMONw. format abbreviates the month's name to three characters.

Example

The example table uses the input value of 19537, which is the SAS date value that corresponds to June 28,2013.

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

See Also

Formats:

- [“DATEw. Format” on page 98](#)
- [“MMYYw. Format” on page 131](#)

YYQw. Format

Writes SAS date values in the form [yy]yyQq, where Q is the separator, the year appears as either 2 or 4 digits, and q is the quarter of the year.

Category:	Date and Time
Alignment:	Right

Syntax

YYQw.

Arguments

w specifies the width of the output field.

Default	6
Range	4–32
Interaction	When w has a value of 4 or 5, the date appears with only the last two digits of the year. When w is 6 or more, the date appears with a four-digit year.

Details

The YYQ_w. format writes SAS date values in the form [yy]yyQ_q, where

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

Q
is the character separator.

q
is an integer (1, 2, 3, or 4) that represents the quarter of the year.

Example

The following examples use the input value of 19537, which is the SAS date value that corresponds to June 28, 2013.

Statements	Results
	-----1-----
select put(19537,yyq4.);	13Q2
select put(19537,yyq5.);	13Q2
select put(19537,yyq.);	2013Q2
select put(19537,yyq6.);	2013Q2
select put(19537,yyq10.);	2013Q2

See Also

Formats:

- [“DATEw. Format” on page 98](#)
- [“YYQ_{xw}. Format” on page 168](#)
- [“YYQRw. Format” on page 170](#)
- [“YYQR_{xw}. Format” on page 171](#)
- [“YYQZw. Format” on page 173](#)

YYQ_{xw}. Format

Writes SAS date values in the form [yy]yyq or [yy]yy-q. The x in the format name is a character that represents the special character that separates the year and the quarter of the year. This character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits.

Category: Date and Time

Alignment: Right

Syntax

YYQ_{xw}.

Arguments

- x**
identifies a separator or specifies that no separator appear between the year and the quarter. Valid values for *x* are any of the following:
- C
separates with a colon
 - D
separates with a hyphen
 - N
indicates no separator
 - P
separates with a period
 - S
separates with a forward slash.

w
specifies the width of the output field.

Default	6
Range	4–32
Interactions	<p>When <i>x</i> is set to <i>N</i>, no separator is specified. The width range is then 3–32, and the default changes to 5.</p> <p>When <i>w</i> has a value of 4 or 5, the date appears with only the last two digits of the year. When <i>w</i> is 6 or more, the date appears with a four-digit year.</p> <p>When <i>x</i> has a value of <i>N</i> and <i>w</i> has a value of 3 or 4, the date appears with only the last two digits of the year. When <i>x</i> has a value of <i>N</i> and <i>w</i> is 5 or more, the date appears with a four-digit year.</p>

Details

The YYQ_{xw}. format writes SAS date values in one of the following forms:

[yy]yyq
[yy]yyxq

where

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

X
is a specified separator.

q
is an integer (1, 2, 3, or 4) that represents the quarter of the year.

Example

The following examples use the input value of 19537, which is the SAS date value that corresponds to July 28, 2013.

Statements	Results
	-----+-----1-----+
select put(19537,yyqc4.);	13:2
select put(19537,yyqd.);	2013-2
select put(19537,yyqn3.);	132
select put(19537,yyqp6.);	2013.2
select put(19537,yyqs8.);	2013/2

See Also

Formats:

- [“DATEw. Format” on page 98](#)
- [“YYQw. Format” on page 167](#)
- [“YYQRw. Format” on page 170](#)
- [“YYQRxw. Format” on page 171](#)
- [“YYQZw. Format” on page 173](#)

YYQRw. Format

Writes SAS date values in the form [yy]yyQqr, where Q is the separator, the year appears as either 2 or 4 digits, and qr is the quarter of the year expressed in roman numerals.

Category: Date and Time

Alignment: Right

Syntax

YYQR^w.

Arguments

^w specifies the width of the output field.

Default 8

Range 6–32

Interaction When the value of *w* is too small to write a four-digit year, the date appears with only the last two digits of the year.

Details

The YYQR*w*. format writes SAS date values in the form [yy]yyQ*qr*, where

<yy>yy

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

Q

is the character separator.

qr

is a roman numeral (I, II, III, or IV) that represents the quarter of the year.

Example

The following examples use the input value of 19537, which is the SAS date value that corresponds to June 28, 2013.

Statements	Results
	----+----1----
select put(19537,yyqr6.);	13QII
select put(19537,yyqr7.);	2013QII
select put(19537,yyqr.);	2013QII
select put(19537,yyqr8.);	2013QII
select put(19537,yyqr10.);	2013QII

See Also

Formats:

- [“YYQw. Format” on page 167](#)
- [“YYQRxw. Format” on page 171](#)

YYQRxw. Format

Writes date values in the form [yy]yy*qr* or [yy]yy-*qr*. The *x* in the format name is a character that represents the special character that separates the year and the quarter of the year. This character can be a hyphen (-), period (.), blank character, slash (/), colon (:), or no separator. The year can be either 2 or 4 digits and *qr* is the quarter of the year in roman numerals.

Category: Date and Time

Alignment: Right

Syntax

YYQR_{xw}.

Arguments

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

- C
separates with a colon
- D
separates with a hyphen
- N
indicates no separator
- P
separates with a period
- S
separates with a forward slash.

w
specifies the width of the output field.

Default	8
Range	6–32
Interactions	When <i>x</i> is set to <i>N</i> , no separator is specified. The width range is then 5–32, and the default changes to 7. When the value of <i>w</i> is too small to write a four-digit year, the date appears with only the last two digits of the year.

Details

The YYQR_{xw}. format writes SAS date values in one of the following forms:

[yy]yyqr
[yy]yyxqr

where

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

X
is a specified separator.

qr
is a roman numeral (I, II, III, or IV) that represents the quarter of the year.

Example

The following examples use the input value of 19721 which is the SAS date value that corresponds to December 29, 2013.

Statements	Results
	----+----1----+
<code>select put(19721,yyqrc6.);</code>	13:IV
<code>select put(19721,yyqrd.);</code>	2013-IV
<code>select put(197214,yyqrn5.);</code>	13IV
<code>select put(19721,yyqrp8.);</code>	2013.IV
<code>select put(19721,yyqrs10.);</code>	2013/IV

See Also

Formats:

- [“YYQxw. Format” on page 168](#)
- [“YYQRw. Format” on page 170](#)

YYQZw. Format

Writes SAS date values in the form `[yy]yyqq`. The year appears as 2 or 4 digits, and `qq` is the quarter of the year.

Category: Date and Time

Alignment: Right

Syntax

YYQZ*w*.

Arguments

Z

specifies that no separator appear between the year and the quarter.

w

specifies the width of the output field.

Default 4

Note 6

Details

The YYQZw. format writes SAS date values in the form `[yy]yyqq` where

`[yy]yy`

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

Z
specifies that there is no separator.

qq
is an integer (01, 02, 03, or 04) that represents the quarter of the year.

Example

The following examples use the input value of 19537, which is the SAS date value that corresponds to June 28, 2013.

Statements	Results
	-----1-----
select put (19537,yyqz6.);	201302
select put (19537,yyqz4.);	1302

See Also

Formats:

- [“DATEw. Format” on page 98](#)
- [“QTRw. Format” on page 143](#)
- [“YYQw. Format” on page 167](#)

Zw.d Format

Writes standard numeric data with leading 0s.

Category: Numeric

Alignment: Right

Syntax

Zw.*[d]*

Arguments

w
specifies the width of the output field.

Default 1

Range 1–32

Tip Allow enough space to write the value, the decimal point, and a minus sign, if necessary.

d

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

Default 0

Range 0–31

Tip If *d* is 0 or you omit *d*, *Zw.d* writes the value without a decimal point.

Details

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

The *Zw.d* format rounds to the nearest number that will fit in the output field. If *w.d* is too large to fit, SAS might shift the decimal to the BEST*w*. format. The *Zw.d* format writes negative numbers with leading minus signs. In addition, it right aligns before writing and pads the output with leading zeros.

Comparisons

The *Zw.d* format is similar to the *w.d* format except that *Zw.d* pads right-aligned output with 0s instead of blanks.

Example

Statement	Result
	-----1
<code>select put(1350,z8.);</code>	00001350

See Also

Formats:

- [“w.d Format” on page 153](#)

Chapter 5

FedSQL Functions

Overview of FedSQL Functions	179
General Function Syntax	180
Aggregate Function Syntax	180
Non-Aggregate Function Syntax	180
Using FedSQL Functions	181
Restrictions on Function Arguments	181
Using DS2 Packages in Expressions	181
Aggregate Functions	182
Overview of Aggregate Functions	182
Calling Base SAS Functions Instead of FedSQL Aggregate Functions	183
Function Categories	183
FEDSQL Functions by Category	184
Dictionary	190
ABS Function	190
ACOS Function	191
ASIN Function	192
ATAN Function	193
ATAN2 Function	194
AVG Function	196
BAND Function	197
BETA Function	198
BLACKCLPRC Function	199
BLACKPTPRC Function	201
BLKSHCLPRC Function	203
BLKSHPTPRC Function	205
BLSHIFT Function	207
BOR Function	208
BRSHIFT Function	209
BXOR Function	210
CAST Function	210
CEIL Function	212
CEILZ Function	213
CHARACTER_LENGTH Function	215
COALESCE Function	216
COALESCEC Function	217
COS Function	218
COSH Function	219
COT Function	220

COUNT Function	221
CSS Function	222
CURRENT_DATE Function	223
CURRENT_LOCALE Function	224
CURRENT_TIME Function	225
CURRENT_TIME_GMT Function	225
CURRENT_TIMESTAMP Function	226
CURRENT_TIMESTAMP_GMT Function	227
CV Function	228
DATE Function	228
DATEJUL Function	229
DATEPART Function	230
DAY Function	231
DEGREES Function	232
E Function	233
EXP Function	234
FLOOR Function	235
FLOORZ Function	236
GAMMA Function	237
GCD Function	238
GEOMEAN Function	239
GEOMEANZ Function	241
HARMEAN Function	242
HARMEANZ Function	243
HOURL Function	245
IFNULL Function	246
IQR Function	247
JULDATE Function	248
JULDATE7 Function	250
LARGEST Function	251
KURTOSIS Function	252
LCM Function	253
LOG Function	254
LOG2 Function	255
LOG10 Function	256
LOGBETA Function	257
LOWCASE Function	258
MAD Function	259
MAKEDATE Function	260
MAKETIME Function	260
MAKETIMESTAMP Function	261
MARGRCLPRC Function	263
MARGRPTPRC Function	265
MAX Function	267
MEAN Function	269
MEDIAN Function	270
MIN Function	271
MINUTE Function	272
MOD Function	273
MONTH Function	275
NMISS Function	276
OCTET_LENGTH Function	277
ORDINAL Function	278
PCTL Function	279
PI Function	281
POWER Function	281

PROBBNML Function	282
PROBBNRM Function	283
PROBCHI Function	284
PROBT Function	285
PUT Function	286
QTR Function	288
QUOTE Function	289
RADIANS Function	290
RANGE Function	291
REPEAT Function	292
REVERSE Function	293
RMS Function	294
SECOND Function	295
SIGN Function	296
SIN Function	297
SINH Function	298
SKEWNESS Function	299
SMALLEST Function	300
SQRT Function	301
STD Function	302
STDDEV Function	303
STDERR Function	304
STUDENTS_T Function	305
SUBSTRING Function	307
SUM Function	308
TAN Function	309
TANH Function	310
TIMEPART Function	311
TODAY Function	312
TRIM Function	313
UPCASE Function	314
USS Function	315
VARIANCE Function	316
WEEKDAY Function	317
YEAR Function	318
YYQ Function	319

Overview of FedSQL Functions

A FedSQL function performs a computation on FedSQL expressions and returns either a single value or a set of values if the FedSQL expression is a column. Functions that perform a computation on a column are *aggregate functions*. In other SQL environments, aggregate functions are also known as set functions.

FedSQL functions can be used in FedSQL applications or in DS2 programs where embedded FedSQL is part of a DS2 application.

For more information, see [“FedSQL Expressions” on page 42](#) and the “SET Statement” in *SAS DS2 Language Reference*.

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

General Function Syntax

Aggregate Function Syntax

The syntax for an aggregate function is

aggregate-function (*expression*)

aggregate-function

specifies the name of the function.

expression

specifies an expression that evaluates to a column name. The expression can include one of the following forms:

table.column

is a qualified column that identifies the table that is being processed by the FedSQL statement followed by a period and the name of the column.

Example `select avg(densities.density) from densities;`

column

is an unqualified column name that has an implicit table qualifier. The implicit table qualifier is the table that FedSQL is processing.

Example `select avg(density) from densities;`

Note: You can specify more than one argument for some aggregate functions. If you do, the Base SAS function is used and the functions can no longer be considered an aggregate function. For more information, see [“Calling Base SAS Functions Instead of FedSQL Aggregate Functions”](#) on page 183.

Non-Aggregate Function Syntax

The syntax of a function is

function (*argument* [, ...*argument*])

function

specifies the name of the function.

argument

can be a variable name, constant, or any FedSQL expression, including another function. The number and type of arguments that FedSQL allows are described with individual functions. Separate multiple arguments by a comma.

Examples `select sqrt(1500);`

`select ifnull(AvgHigh, 0) from worldtemps;`

Using FedSQL Functions

Restrictions on Function Arguments

If the value of an argument is invalid, FedSQL sets the result to a null or missing value. Here are some common restrictions on function arguments:

- Some functions require that their arguments be restricted within a certain range. For example, the argument of the LOG function must be greater than 0.
- Most functions do not permit nulls or missing values as arguments. Exceptions include some of the descriptive statistics functions and the IFNULL function.
- In general, the allowed range of the arguments is platform-dependent, such as with the EXP function.

Using DS2 Packages in Expressions

You can invoke a DS2 package method as a function in a FedSQL SELECT statement. The syntax for the expression is shown here.

```
[catalog.] [schema.] package.method (argument-1 [... argument-n])
```

The following restrictions apply when you use DS2 packages in expressions in FedSQL:

- The method parameters must be a DOUBLE, an integer of any size, or a string.
- The method must return a double, an integer of any size, or a string.
- The method must return a value.
- You must supply a package name.
- The types and ordering of the arguments in the FedSQL SELECT statement must match the types and ordering of the parameters in the DS2 package method.

In the following example, the method **BAR** is created in DS2 and then used in a FedSQL SELECT statement call as a function.

```
/* create table */
data dataset;
  x=1; y=1; z=1; output;
  x=1; y=1; z=2; output;
  x=2; y=2; z=2; output;
  x=2; y=2; z=8; output;
  x=2; y=3; z=13; output;
run;

/* DS2 code */
proc ds2;
  package pkgA;
    method bar(double x, double y) returns double;
      return x*x + y*y;
    end;
  endpackage;
run;
```

```
quit;

/* fedsql code */
proc fedsql;
  select pkga.bar(1,2) as five,
         cot(radians(45)) as one,
         degrees(pi()) as one_eighty,
         power(3,4) as eighty_one,
         sign(-42) as minus1,
         sign(0) as zero,
         sign(42) as plus1;
  select * from dataset where pkga.bar(x,y) = z;
quit;
```

The output from the FedSQL SELECT statement would be as follows.

FIVE	ONE	ONE_EIGHTY	EIGHTY_ONE	MINUS1	ZERO	PLUS1
5	1	180	81	-1	0	1

x	y	z
1	1	2
2	2	8
2	3	13

For more information, see “Package Method Expression” in Chapter 13 of *SAS DS2 Language Reference*.

Aggregate Functions

Overview of Aggregate Functions

FedSQL aggregate functions operate on all values for an expression in a table column and return a single result. If the aggregate function is processed in a GROUP BY statement, the aggregate function returns a single result for each group. Null values and SAS missing values are not considered in the operation, except for the COUNT(*) syntax of the COUNT function. The table column that you specify in the function can be any FedSQL expression that evaluates to a column name.

The following are FedSQL aggregate functions:

- “AVG Function” on page 196
- “COUNT Function” on page 221
- “CSS Function” on page 222
- “KURTOSIS Function” on page 252

- “MAX Function” on page 267
- “MIN Function” on page 271
- “NMISS Function” on page 276
- “PROBT Function” on page 285
- “RANGE Function” on page 291
- “SKEWNESS Function” on page 299
- “STD Function” on page 302
- “STDDEV Function” on page 303
- “STDERR Function” on page 304
- “STUDENTS_T Function” on page 305
- “SUM Function” on page 308
- “USS Function” on page 315
- “VARIANCE Function” on page 316

Using the table “WorldTemps” on page 542, the following aggregate function examples operate on the AvgLow table column:

```
/* Get the average of the average low temperatures */
select avg(AvgLow) as AvgTemp from worldtemps;

/* Get the number of different average low temperatures */
/* and group them by the average low temperature */
select AvgLow, count(AvgLow) from worldtemps group by AvgLow;

/* Get the highest average low temperature */
select max(AvgLow) from worldtemps;
```

Calling Base SAS Functions Instead of FedSQL Aggregate Functions

If multiple columns are supplied as arguments to an aggregate function and there is a like-named Base SAS function, the Base SAS function is used. The statistic that is calculated for those arguments is for the current row. The function is no longer considered to be an aggregate function. Some examples are the MIN, MAX, and SUM functions.

If multiple arguments are supplied to an aggregate function and there is no like-named Base SAS function, an error is returned. An example is the AVG function.

Function Categories

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

Aggregate

operates on the values in a table column

Bitwise Logical Operations

operates on one or more bit patterns or binary numbers at the level of their individual bits

Character

operates on character SQL expressions

Date and Time

operates on date and time SQL expressions

Descriptive Statistics

operates on values that measure central tendency, variation among values, and the shape of distribution values

Financial

calculates financial values such as interest, periodic payments, depreciation, and prices for European options on stocks.

Mathematical

operates on values to perform general mathematical calculations

Probability

returns probability calculations.

Special

operates on null values and SAS missing values

Trigonometric

operates on values to perform trigonometric calculations

Truncation

truncates numeric values and returns numeric values, often using fuzzing

FEDSQL Functions by Category

Category	Language Elements	Description
Aggregate	AVG Function (p. 196)	Returns the average of all values in a column.
	COUNT Function (p. 221)	Returns the number of rows retrieved by a SELECT statement for a specified table.
	CSS Function (p. 222)	Returns the corrected sum of squares of all values in an expression.
	KURTOSIS Function (p. 252)	Returns the kurtosis of all values in an expression.
	MAX Function (p. 267)	Returns the maximum value in a column.
	MIN Function (p. 271)	Returns the minimum value in an expression.
	NMISS Function (p. 276)	Returns the number of null values or SAS missing values in an expression.
	PROBT Function (p. 285)	Returns the probability from a t distribution of the values in an expression.

Category	Language Elements	Description
	RANGE Function (p. 291)	Returns the range between values in an expression.
	SKEWNESS Function (p. 299)	Returns the skewness of all values in an expression.
	STD Function (p. 302)	Returns the standard deviation.
	STDDEV Function (p. 303)	Returns the statistical standard deviation of all values in an expression.
	STDERR Function (p. 304)	Returns the statistical standard error of all values in an expression.
	STUDENTS_T Function (p. 305)	Returns the Student's t distribution of the values in an expression.
	SUM Function (p. 308)	Returns the sum of all the values in an expression.
	USS Function (p. 315)	Returns the uncorrected sum of squares of all the values in an expression.
	VARIANCE Function (p. 316)	Returns the measure of the dispersion of all values in an expression.
Bitwise Logical Operations	BAND Function (p. 197)	Returns the bitwise logical AND of two arguments.
	BLSHIFT Function (p. 207)	Returns the bitwise logical left shift of two arguments.
	BOR Function (p. 208)	Returns the bitwise logical OR of two arguments.
	BRSHIFT Function (p. 209)	Returns the bitwise logical right shift of two arguments.
	BXOR Function (p. 210)	Returns the bitwise logical EXCLUSIVE OR of two arguments.
Character	CHARACTER_LENGTH Function (p. 215)	Returns the number of characters in a string of any data type.
	COALESCEC Function (p. 217)	Returns the first non-null or nonmissing value from a list of character arguments.
	CURRENT_LOCALE Function (p. 224)	Returns the five character name of the current locale.
	LOWCASE Function (p. 258)	Converts all letters in a character expression to lowercase.
	OCTET_LENGTH Function (p. 277)	Returns the number of bytes in a string of any data type.
	QUOTE Function (p. 289)	Adds double quotation marks to a character value.
	REPEAT Function (p. 292)	Repeats a character expression.
	REVERSE Function (p. 293)	Reverses a character expression.

Category	Language Elements	Description
Date and Time	SUBSTRING Function (p. 307)	Extracts a substring from a character string.
	TRIM Function (p. 313)	Removes leading characters, trailing characters, or both from a character string.
	UPCASE Function (p. 314)	Converts all letters in an argument to uppercase.
	CURRENT_DATE Function (p. 223)	Returns the current date for the time zone.
	CURRENT_TIME Function (p. 225)	Returns the current time for your time zone.
	CURRENT_TIME_GMT Function (p. 225)	Returns the current GMT time.
	CURRENT_TIMESTAMP Function (p. 226)	Returns the date and time for your time zone.
	CURRENT_TIMESTAMP_GMT Function (p. 227)	Returns the current GMT date and time.
	DATE Function (p. 228)	Returns the current date as a SAS date value.
	DATEJUL Function (p. 229)	Converts a Julian date to a SAS date value.
	DATEPART Function (p. 230)	Returns the date as year, month, and day.
	DAY Function (p. 231)	Returns the numeric day of the month from a date or datetime value.
	HOURL Function (p. 245)	Returns the hour from a time or datetime value.
	JULDATE Function (p. 248)	Returns the Julian date from a SAS date value.
	JULDATE7 Function (p. 250)	Returns a seven-digit Julian date from a SAS date value.
	MAKEDATE Function (p. 260)	Returns the date as year, month, and day.
	MAKETIME Function (p. 260)	Returns the time as hours, minutes, and seconds.
	MAKETIMESTAMP Function (p. 261)	Returns the timestamp.
	MINUTE Function (p. 272)	Returns the minute from a time or datetime value.
	MONTH Function (p. 275)	Returns the numeric month from a date or datetime value.
	QTR Function (p. 288)	Returns the quarter of the year from a SAS date value.
	SECOND Function (p. 295)	Returns the second from a time or datetime value.

Category	Language Elements	Description
Descriptive Statistics	TIMEPART Function (p. 311)	Returns the time as hours, minutes, and seconds.
	TODAY Function (p. 312)	Returns the current date as a numeric SAS date value.
	WEEKDAY Function (p. 317)	From a SAS date value, returns an integer that corresponds to the day of the week.
	YEAR Function (p. 318)	Returns the year from a date or datetime value.
	YYQ Function (p. 319)	Returns a SAS date value from year and quarter year values.
	AVG Function (p. 196)	Returns the average of all values in a column.
	CSS Function (p. 222)	Returns the corrected sum of squares of all values in an expression.
	CV Function (p. 228)	Returns the coefficient of variation.
	GEOMEAN Function (p. 239)	Returns the geometric mean.
	GEOMEANZ Function (p. 241)	Returns the geometric mean, using zero fuzzing.
	HARMEAN Function (p. 242)	Returns the harmonic mean.
	HARMEANZ Function (p. 243)	Returns the harmonic mean, using zero fuzzing.
	IQR Function (p. 247)	Returns the interquartile range.
	LARGEST Function (p. 251)	Returns the kth largest non-null or nonmissing value.
	KURTOSIS Function (p. 252)	Returns the kurtosis of all values in an expression.
	MAD Function (p. 259)	Returns the median absolute deviation from the median.
	MAX Function (p. 267)	Returns the maximum value in a column.
	MEAN Function (p. 269)	Returns the arithmetic mean (average) of the non-null or nonmissing arguments.
	MEDIAN Function (p. 270)	Returns the median value.
	MIN Function (p. 271)	Returns the minimum value in an expression.
	ORDINAL Function (p. 278)	Orders a list of values, and returns a value that is based on a position in the list.
	PCTL Function (p. 279)	Returns the percentile that corresponds to the percentage.
	PROBT Function (p. 285)	Returns the probability from a t distribution of the values in an expression.

Category	Language Elements	Description
	RANGE Function (p. 291)	Returns the range between values in an expression.
	RMS Function (p. 294)	Returns the root mean square.
	SKEWNESS Function (p. 299)	Returns the skewness of all values in an expression.
	SMALLEST Function (p. 300)	Returns the kth smallest non-null or nonmissing value.
	STD Function (p. 302)	Returns the standard deviation.
	STDDEV Function (p. 303)	Returns the statistical standard deviation of all values in an expression.
	STDERR Function (p. 304)	Returns the statistical standard error of all values in an expression.
	STUDENTS_T Function (p. 305)	Returns the Student's t distribution of the values in an expression.
	SUM Function (p. 308)	Returns the sum of all the values in an expression.
	USS Function (p. 315)	Returns the uncorrected sum of squares of all the values in an expression.
	VARIANCE Function (p. 316)	Returns the measure of the dispersion of all values in an expression.
Financial	BLACKCLPRC Function (p. 199)	Calculates call prices for European options on futures, based on the Black model.
	BLACKPTPRC Function (p. 201)	Calculates put prices for European options on futures, based on the Black model.
	BLKSHCLPRC Function (p. 203)	Calculates call prices for European options on stocks, based on the Black-Scholes model.
	BLKSHPTPRC Function (p. 205)	Calculates put prices for European options on stocks, based on the Black-Scholes model.
	MARGRCLPRC Function (p. 263)	Calculates call prices for European options on stocks, based on the Margrabe model.
	MARGRPTPRC Function (p. 265)	Calculates put prices for European options on stocks, based on the Margrabe model.
Mathematical	ABS Function (p. 190)	Returns the absolute value of a numeric value expression.
	BETA Function (p. 198)	Returns the value of the beta function.
	COALESCE Function (p. 216)	Returns the first non-null or nonmissing value from a list of numeric arguments.
	E Function (p. 233)	Returns the natural logarithm, e.

Category	Language Elements	Description
	EXP Function (p. 234)	Returns the value of the e constant raised to a specified power.
	GAMMA Function (p. 237)	Returns the value of the gamma function.
	GCD Function (p. 238)	Returns the greatest common divisor for a set of integers.
	LCM Function (p. 253)	Returns the least common multiple for a set of integers.
	LOG Function (p. 254)	Returns the natural logarithm (base e) of a numeric value expression.
	LOG2 Function (p. 255)	Returns the base-2 logarithm of a numeric value expression.
	LOG10 Function (p. 256)	Returns the base-10 logarithm of a numeric value expression.
	LOGBETA Function (p. 257)	Returns the logarithm of the beta function.
	MOD Function (p. 273)	Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid most unexpected floating-point results.
	PI Function (p. 281)	Returns the constant value of PI as a floating-point value.
	POWER Function (p. 281)	Returns the value of a numeric value expression raised to a specified power.
	SIGN Function (p. 296)	Returns a number that indicates the sign of a numeric value expression.
	SQRT Function (p. 301)	Returns the square root of a value.
Probability	PROBBNML Function (p. 282)	Returns the probability from a binomial distribution.
	PROBBNRM Function (p. 283)	Returns a probability from a bivariate normal distribution.
	PROBCHI Function (p. 284)	Returns the probability from a chi-square distribution.
Special	CAST Function (p. 210)	Converts a value from one data type to another.
	IFNULL Function (p. 246)	Checks the value of the first expression and, if it is null or a SAS missing value, returns the second expression.
	PUT Function (p. 286)	Returns a value using a specified format.
Trigonometric	ACOS Function (p. 191)	Returns the arccosine in radians.
	ASIN Function (p. 192)	Returns the arcsine in radians.
	ATAN Function (p. 193)	Returns the arctangent in radians.
	ATAN2 Function (p. 194)	Returns the arctangent of the x and y coordinates of a right triangle, in radians.

Category	Language Elements	Description
	COS Function (p. 218)	Returns the cosine in radians.
	COSH Function (p. 219)	Returns the hyperbolic cosine in radians.
	COT Function (p. 220)	Returns the tangent in radians.
	DEGREES Function (p. 232)	Returns the number of degrees for an angle in radians.
	RADIANS Function (p. 290)	Returns the number of radians converted from a numeric degree value.
	SIN Function (p. 297)	Returns the trigonometric sine.
	SINH Function (p. 298)	Returns the hyperbolic sine.
	TAN Function (p. 309)	Returns the tangent.
Truncation	TANH Function (p. 310)	Returns the hyperbolic tangent.
	CEIL Function (p. 212)	Returns the smallest integer greater than or equal to a numeric value expression.
	CEILZ Function (p. 213)	Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing.
	FLOOR Function (p. 235)	Returns the largest integer less than or equal to a numeric value expression.
	FLOORZ Function (p. 236)	Returns the largest integer that is less than or equal to the argument, using zero fuzzing.

Dictionary

ABS Function

Returns the absolute value of a numeric value expression.

Category: Mathematical

Returned data type: The same data type as the expression

Syntax

ABS(*expression*)

Arguments

expression

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

Type BIGINT, DOUBLE, FLOAT, INTEGER, REAL, SMALLINT, TINYINT

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

Details

If *expression* is null, then the ABS function returns null. If the result is a number that does not fit into the range of the argument's data type, the ABS function fails.

Example

The following statements illustrate the ABS function:

Statements	Results
<code>select abs(-345);</code>	345
<code>select abs((3 * 50) / 5)</code>	30

ACOS Function

Returns the arccosine in radians.

Category: Trigonometric

Alias: ARCOS

Returned data type: DOUBLE

Syntax

`ACOS(expression)`

Arguments

expression

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

Range -1 to 1

Data type DOUBLE

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

Details

The ACOS function returns the arccosine (inverse cosine) of the argument. The value that is returned is specified in radians.

Example

The following statements illustrate the ACOS function:

Statements	Results
<code>select acos(1);</code>	0
<code>select arcos(0);</code>	1.570796
<code>select acos(-0.5);</code>	2.094395

See Also

Functions:

- [“ASIN Function” on page 192](#)
- [“COS Function” on page 218](#)
- [“SIN Function” on page 297](#)

ASIN Function

Returns the arcsine in radians.

Category:	Trigonometric
Alias:	ARSIN
Returned data type:	DOUBLE

Syntax

`ASIN(expression)`

Arguments

expression

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

Range	−1 to 1
Data type	DOUBLE
See	“<sql-expression>” on page 339
	“FedSQL Expressions” on page 42

Example

The following statements illustrate the ASIN function:

Statements	Results
<code>select asin(0);</code>	0
<code>select asin(1);</code>	1.570796
<code>select asin(-0.5);</code>	-0.5236

See Also

Functions:

- [“ACOS Function” on page 191](#)
- [“COS Function” on page 218](#)
- [“COSH Function” on page 219](#)
- [“SIN Function” on page 297](#)

ATAN Function

Returns the arctangent in radians.

Category: Trigonometric

Alias: ARTAN

Returned data type: DOUBLE

Syntax

ATAN(*expression*)

Arguments

expression

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

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The ATAN function returns the 2-quadrant arctangent (inverse tangent) of the argument. The value that is returned is the angle (in radians) whose tangent is *x* and whose value

ranges from $-\pi/2$ to $\pi/2$. If the argument is missing, then ATAN returns a missing value.

Comparisons

The ATAN function is similar to the ATAN2 function except that ATAN2 calculates the arc tangent of the angle from the values of two arguments rather than from one argument.

Example

The following statements illustrate the ATAN function:

Statements	Results
<code>select atan(0);</code>	0
<code>select atan(1);</code>	0.785398
<code>select atan(-9.0);</code>	-1.46014

See Also

Functions:

- [“ATAN2 Function” on page 194](#)
- [“COT Function” on page 220](#)
- [“TAN Function” on page 309](#)
- [“TANH Function” on page 310](#)

ATAN2 Function

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

Category: Trigonometric
Returned data type: DOUBLE

Syntax

`ATAN2(expression-1, expression-2)`

Arguments

expression-1

specifies any valid SQL expression that evaluates to a numeric value. *expression-1* specifies the x coordinate of the end of the hypotenuse of a right triangle.

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

expression-2

specifies any valid SQL expression that evaluates to a numeric value. *expression-2* specifies the y coordinate of the end of the hypotenuse of a right triangle.

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The ATAN2 function returns the arctangent (inverse tangent) of two numeric variables. The result of this function is similar to the result of calculating the arctangent of *expression-1* / *expression-2*, except that the signs of both arguments are used to determine the quadrant of the result. If either of the arguments in ATAN2 is missing, then ATAN2 returns either a null or a SAS missing value.

Comparisons

The ATAN2 function is similar to the ATAN function except that ATAN calculates the arctangent of the angle from the value of one argument rather than from two arguments.

Example

The following statements illustrate the ATAN2 function:

Statements	Results
<code>select atan2(-1, 0.5);</code>	-1.10715
<code>select atan2(6,8);</code>	0.643501
<code>select atan2(5,-3);</code>	2.111216

See Also

- [“How FedSQL Processes Nulls and SAS Missing Values” on page 18](#)

Functions:

- [“ATAN Function” on page 193](#)
- [“TAN Function” on page 309](#)
- [“TANH Function” on page 310](#)

AVG Function

Returns the average of all values in a column.

Categories: Aggregate
Descriptive Statistics

Alias: MEAN

Returned data type: DOUBLE

Syntax

AVG(*expression*)

Arguments

expression
specifies any valid SQL expression.

Data type BIGINT, DOUBLE, FLOAT, INTEGER, REAL, SMALLINT, TINYINT

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The AVG function adds the values of all the rows in the specified column and divides the result by the number of rows. Null values and SAS missing values are ignored and are not included in the computation.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Example

Table: [DENSITIES on page 539](#)

The following statements illustrate the AVG function:

Statements	Results
select avg(density) from densities;	172.8324
select avg(population) from densities;	12277544

See Also

Functions:

- [“SUM Function” on page 308](#)

Clauses:

- [“SELECT Clause” on page 389](#)
- [“GROUP BY Clause” on page 400](#)
- [“HAVING Clause” on page 401](#)

BAND Function

Returns the bitwise logical AND of two arguments.

Category: Bitwise Logical Operations

Returned data type: DOUBLE

Syntax

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

Arguments

expression-1, *expression-2*

specifies any valid expression that evaluates to a numeric value.

Range between 0 and $(2^{32})-1$ inclusive

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Example

The following statements illustrate the BAND function:

Statements	Results
<code>select band(9,11);</code>	9
<code>select band(15,5);</code>	5

BETA Function

Returns the value of the beta function.

Category:	Mathematical
Returned data type:	DOUBLE

Syntax

BETA(*a*, *b*)

Arguments

a
is the first shape parameter.

Range	<i>a</i> > 0
Data type	DOUBLE

b
is the second shape parameter.

Range	<i>b</i> > 0
Data type	DOUBLE

Details

The BETA function is mathematically given by this equation:

$$\beta(a, b) = \int_0^1 x^{a-1} (1-x)^{b-1} dx$$

Note the following:

$$\beta(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

In the previous equation, $\Gamma(.)$ is the gamma function.

If the expression cannot be computed, BETA returns a missing value.

Example

The following statements illustrate the BETA function:

Statements	Results
<pre>select beta(5,3);</pre>	0.00952380952381
<pre>select beta(15,45);</pre>	1.6710294365E-15

See Also

Functions:

- [“LOGBETA Function” on page 257](#)

BLACKCLPRC Function

Calculates call prices for European options on futures, based on the Black model.

Category: Financial

Returned data type: DOUBLE

Syntax

BLACKCLPRC(*E*, *t*, *F*, *r*, *sigma*)

Arguments

E

is a nonmissing, positive value that specifies exercise price.

Requirement Specify *E* and *F* in the same units.

Data type DOUBLE

t

is a nonmissing value that specifies time to maturity, in years.

Data type DOUBLE

F

is a nonmissing, positive value that specifies future price.

Requirement Specify *F* and *E* in the same units.

Data type DOUBLE

r

is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

Data type DOUBLE

sigma

is a nonmissing, positive fraction that specifies the volatility (the square root of the variance of *r*).

Data type DOUBLE

Details

The BLACKCLPRC function calculates call prices for European options on futures, based on the Black model. The function is based on the following relationship:

$$CALL = e^{-rt}(FN(d_1) - EN(d_2))$$

Arguments

F

specifies future price.

N

specifies the cumulative normal density function.

E

specifies the exercise price of the option.

r

specifies the risk-free interest rate, which is an annual rate that is expressed in terms of continuous compounding.

t

specifies the time to expiration, in years.

$$d_1 = \frac{\left(\ln\left(\frac{F}{E}\right) + \left(\frac{\sigma^2}{2}\right)t \right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

The following arguments apply to the preceding equation:

σ

specifies the volatility of the underlying asset.

σ^2

specifies the variance of the rate of return.

For the special case of $t=0$, the following equation is true:

$$CALL = \max((F - E), 0)$$

For information about the basics of pricing, see “Using Pricing Functions” in Chapter 1 of *SAS Functions and CALL Routines: Reference*.

Comparisons

The BLACKCLPRC function calculates call prices for European options on futures, based on the Black model. The BLACKPTPRC function calculates put prices for European options on futures, based on the Black model. These functions return a scalar value.

Example

The following statements illustrate the BLACKCLPRC function:

Statements	Results
-----1-----2--	

Statements	Results
<code>select blackclprc(50,.25,48,.05,.25);</code>	1.55130142723117
<code>select blackclprc(9,1/12,10,.05,.2);</code>	1

See Also

Functions:

- [“BLACKPTPRC Function” on page 201](#)

BLACKPTPRC Function

Calculates put prices for European options on futures, based on the Black model.

Category: Financial

Returned data type: DOUBLE

Syntax

BLACKPTPRC(*E*, *t*, *F*, *r*, *sigma*)

Arguments

E

is a nonmissing, positive value that specifies exercise price.

Requirement Specify *E* and *F* in the same units.

Data type DOUBLE

t

is a nonmissing value that specifies time to maturity, in years.

Data type DOUBLE

F

is a nonmissing, positive value that specifies future price.

Requirement Specify *F* and *E* in the same units.

Data type DOUBLE

r

is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

Data type DOUBLE

sigma

is a nonmissing, positive fraction that specifies the volatility (the square root of the variance of r).

Data type DOUBLE

Details

The BLACKPTPRC function calculates put prices for European options on futures, based on the Black model. The function is based on the following relationship:

$$PUT = CALL + e^{-rt}(E - F)$$

Arguments

E

specifies the exercise price of the option.

r

specifies the risk-free interest rate, which is an annual rate that is expressed in terms of continuous compounding.

t

specifies the time to expiration, in years.

F

specifies future price.

$$d_1 = \frac{\left(\ln \left(\frac{F}{E} \right) + \left(\frac{\sigma^2}{2} \right) t \right)}{\sigma \sqrt{t}}$$

$$d_2 = d_1 - \sigma \sqrt{t}$$

The following arguments apply to the preceding equation:

σ

specifies the volatility of the underlying asset.

σ^2

specifies the variance of the rate of return.

For the special case of $t=0$, the following equation is true:

$$PUT = \max((E - F), 0)$$

For information about the basics of pricing, see “Using Pricing Functions” in Chapter 1 of *SAS Functions and CALL Routines: Reference*.

Comparisons

The BLACKPTPRC function calculates put prices for European options on futures, based on the Black model. The BLACKCLPRC function calculates call prices for European options on futures, based on the Black model. These functions return a scalar value.

Example

The following statements illustrate the BLACKPTPRC function:

Statements	Results
	-----1-----2--
<code>select blackptprc(298,.25,350,.06,.25);</code>	1.85980563934967
<code>select blackptprc(145,.5,170,.05,.2);</code>	1.41234979911583

See Also

Functions:

- [“BLACKCLPRC Function” on page 199](#)

BLKSHCLPRC Function

Calculates call prices for European options on stocks, based on the Black-Scholes model.

Category: Financial

Returned data type: DOUBLE

Syntax

BLKSHCLPRC(*E*, *t*, *S*, *r*, *sigma*)

Arguments

E

is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

Data type DOUBLE

t

is a nonmissing value that specifies the time to maturity, in years.

Data type INTEGER

S

is a nonmissing, positive value that specifies the share price.

Requirement Specify *S* and *E* in the same units.

Data type DOUBLE

r

is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

Data type DOUBLE

sigma

is a nonmissing, positive fraction that specifies the volatility of the underlying asset.

Data type DOUBLE

Details

The BLKSHCLPRC function calculates the call prices for European options on stocks, based on the Black-Scholes model. The function is based on the following relationship:

$$CALL = SN(d_1) - EN(d_2)e^{-rt}$$

Arguments

S

is a nonmissing, positive value that specifies the share price.

N

specifies the cumulative normal density function.

E

is a nonmissing, positive value that specifies the exercise price of the option.

$$d_1 = \frac{\left(\ln\left(\frac{S}{E}\right) + \left(r + \frac{\sigma^2}{2}\right)t \right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

The following arguments apply to the preceding equation:

t

specifies the time to expiration, in years.

r

specifies the risk-free interest rate, which is an annual rate that is expressed in terms of continuous compounding.

σ

specifies the volatility (the square root of the variance).

σ²

specifies the variance of the rate of return.

For the special case of *t*=0, the following equation is true:

$$CALL = \max((S - E), 0)$$

For information about the basics of pricing, see “Using Pricing Functions” in Chapter 1 of *SAS Functions and CALL Routines: Reference*.

Comparisons

The BLKSHCLPRC function calculates the call prices for European options on stocks, based on the Black-Scholes model. The BLKSHPTPRC function calculates the put prices for European options on stocks, based on the Black-Scholes model. These functions return a scalar value.

Example

The following statements illustrate the BLKSHCLPRC function:

Statements	Results
	-----1-----2--
<code>select blkshc1prc(50,.25,48,.05,.25);</code>	1.79894201954463
<code>select blkshc1prc(9,1/12,10,.05,.2);</code>	1

See Also

Functions:

- [“BLKSHPTPRC Function” on page 205](#)

BLKSHPTPRC Function

Calculates put prices for European options on stocks, based on the Black-Scholes model.

Category: Financial

Returned data type: DOUBLE

Syntax

BLKSHPTPRC(*E*, *t*, *S*, *r*, *sigma*)

Arguments

E

is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

Data type DOUBLE

t

is a nonmissing value that specifies the time to maturity, in years.

Data type INTEGER

S

is a nonmissing, positive value that specifies the share price.

Requirement Specify *S* and *E* in the same units.

Data type DOUBLE

r

is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

Data type DOUBLE

sigma

is a nonmissing, positive fraction that specifies the volatility of the underlying asset.

Data type DOUBLE

Details

The BLKSHPTPRC function calculates the put prices for European options on stocks, based on the Black-Scholes model. The function is based on the following relationship:

$$PUT = CALL - S + Ee^{-rt}$$

Arguments

S

is a nonmissing, positive value that specifies the share price.

E

is a nonmissing, positive value that specifies the exercise price of the option.

$$d_1 = \frac{\left(\ln \left(\frac{S}{E} \right) + \left(r + \frac{\sigma^2}{2} \right) t \right)}{\sigma \sqrt{t}}$$

$$d_2 = d_1 - \sigma \sqrt{t}$$

The following arguments apply to the preceding equation:

t

specifies the time to expiration, in years.

r

specifies the risk-free interest rate, which is an annual rate that is expressed in terms of continuous compounding.

σ

specifies the volatility (the square root of the variance).

σ²

specifies the variance of the rate of return.

For the special case of *t*=0, the following equation is true:

$$PUT = \max((E - S), 0)$$

For information about the basics of pricing, see “Using Pricing Functions” in Chapter 1 of *SAS Functions and CALL Routines: Reference*.

Comparisons

The BLKSHPTPRC function calculates the put prices for European options on stocks, based on the Black-Scholes model. The BLKSHCLPRC function calculates the call prices for European options on stocks, based on the Black-Scholes model. These functions return a scalar value.

Example

The following statements illustrate the BLKSHPTPRC function:

Statements	Results
	-----1-----2--
<code>select blkshptprc(230,.5,290,.04,.25);</code>	1.56597442946068
<code>select blkshptprc(350,.3,400,.05,.2);</code>	1.64091943067592

See Also

Functions:

- [“BLKSHCLPRC Function” on page 203](#)

BLSHIFT Function

Returns the bitwise logical left shift of two arguments.

Category: Bitwise Logical Operations

Returned data type: DOUBLE

Syntax

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

Arguments

expression-1

specifies any valid expression that evaluates to a numeric value.

Range between 0 and $(2^{32})-1$ inclusive

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

expression-2

specifies any valid expression that evaluates to a numeric value.

Range 0 to 31, inclusive

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Example

The following statements illustrate the BLSHIFT function:

Statements	Results
<code>select blshift(7,2);</code>	28

See Also

Functions:

- [“BRSHIFT Function” on page 209](#)

BOR Function

Returns the bitwise logical OR of two arguments.

Category: Bitwise Logical Operations

Returned data type: DOUBLE

Syntax

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

Arguments

expression-1, *expression-2*

specifies any valid expression that evaluates to a numeric value.

Range between 0 and $(2^{32})-1$ inclusive

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Example

The following statements illustrate the BOR function:

Statements	Results
<code>select bor(4,8);</code>	12

BRSHIFT Function

Returns the bitwise logical right shift of two arguments.

Category: Bitwise Logical Operations

Returned data type: DOUBLE

Syntax

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

Arguments

expression-1

specifies any valid expression that evaluates to a numeric value.

Range	between 0 and $(2^{32})-1$ inclusive
Data type	DOUBLE
See	“<sql-expression>” on page 339 “FedSQL Expressions” on page 42

expression-2

specifies any valid expression that evaluates to a numeric value.

Range	0 to 31, inclusive
Data type	DOUBLE
See	“<sql-expression>” on page 339 “FedSQL Expressions” on page 42

Example

The following statement illustrates the BRSHIFT function:

Statements	Results
<code>select brshift(64,2);</code>	16

See Also

Functions:

- [“BLSHIFT Function” on page 207](#)

BXOR Function

Returns the bitwise logical EXCLUSIVE OR of two arguments.

Category: Bitwise Logical Operations

Returned data type: DOUBLE

Syntax

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

Arguments

expression-1, *expression-2*

specifies any valid expression that evaluates to a numeric value.

Range between 0 and $(2^{32})-1$ inclusive

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Example

The following statement illustrates the BXOR function:

Statements	Results
<code>select bxor(128,64);</code>	192

CAST Function

Converts a value from one data type to another.

Category: Special

Alias: When *expression* is the name of a table column, the operator “::” can be used in the place of the CAST keyword.

Returned data type: The target data type.

Syntax

CAST(*expression AS data-type* [(*length*)])

Arguments

expression

specifies any valid SQL expression.

Data type Expression can resolve to any FedSQL data type supported by the data source.

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

[Appendix 2, “Data Type Reference,” on page 507](#)

data-type

Is the target data type. See “Details” for information about supported data type conversions.

Restriction The target data type must be supported by FedSQL and by the data source.

See [“Data Types” on page 13](#)

[Appendix 2, “Data Type Reference,” on page 507](#)

length

Is an optional integer that specifies the length of the target data type. The length argument is intended for use with character values. It is important when specifying literals. When a literal value is specified, the default value is 0, and the length needs to be at least as long as the number of characters that will be generated by the CAST.

Details

FedSQL performs implicit type conversions as needed for data source support. The CAST function is provided for performing explicit type conversions.

CAST expressions are permitted anywhere expressions are permitted. In addition to converting one data type to another, CAST can be used to change the length of the target type.

The following type conversions are supported:

- A character type can be converted to another character type, a numeric type, and a DATE or TIMESTAMP.
- A numeric type can be converted to CHAR or another numeric type. If the target type cannot represent the non-fractional component without truncation, an exception is raised. If the target numeric cannot represent the fractional component (scale) of the source numeric, then the source is silently truncated to fit into the target. For example, casting 123.4763 as INTEGER yields 123.
- A DATE or TIME value can be converted to a TIMESTAMP and a DOUBLE. TIMESTAMP can also be converted to a DOUBLE. If a DATE is converted to a TIMESTAMP, the TIME component of the resulting TIMESTAMP is always 00:00:00. If a TIME data value is converted to a TIMESTAMP, the DATE component is set to the value of CURRENT_DATE at the time the CAST is executed. If a TIMESTAMP is converted to a DATE, the TIME component is silently truncated. If a TIMESTAMP is converted to a TIME, the DATE component is silently truncated.

Comparisons

The CAST function permanently modifies the data type of the specified input variable. The PUT function affects the output of the query in which it is specified.

Example

Table: [“CustonLine” on page 538](#)

Table: [“Integers” on page 541](#)

The following statements illustrate the CAST function.

Statements	Results
select cast(beginntime as DATE) from custonline;	01SEP2013 02OCT2013 15OCT2013 01NOV2013 01DEC2013 02JAN2013 16JAN2013 01FEB2013 01MAR2013 15MAR2013
select si::integer as int from integers;	32767
select bi::bigint * 2::bigint as bigbang from integers;	ERROR: Numeric value out of range
select cast ('2014-04-10' as DATE);	10APR2014
select cast ('2014-04-10 10:56:49' as TIMESTAMP);	10APR2014:10:56:49
select cast ('10:56:49' as TIME);	10:56:49

CEIL Function

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

Category: Truncation

Returned data type: DECIMAL, DOUBLE, NUMERIC

Syntax

CEIL(*expression*)

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Data type DECIMAL, DOUBLE, NUMERIC

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

Details

If *expression* is null, then the CEILING function returns null. If the result is a number that does not fit into the range of the argument's data type, the CEIL function fails.

If the argument is DECIMAL, the result is DECIMAL. Otherwise, the argument is converted to DOUBLE (if not so already), and the result is DOUBLE.

Comparisons

Unlike the CEILZ function, the CEIL function fuzzes the result. If the argument is within 1E-12 of an integer, the CEIL function fuzzes the result to be equal to that integer. The CEILZ function does not fuzz the result. Therefore, with the CEILZ function, you might get unexpected results.

Example

The following statements illustrate the CEIL function:

Statements	Results
<code>select ceil(-2.4);</code>	-2
<code>select ceil(1+1.e-11);</code>	2
<code>select ceil(-1+1.e-11);</code>	0
<code>select ceil(1+1.e-13);</code>	1

See Also

Functions:

- [“CEILZ Function” on page 213](#)
- [“FLOOR Function” on page 235](#)
- [“FLOORZ Function” on page 236](#)

CEILZ Function

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

Category: Truncation**Returned data type:** DOUBLE

Syntax

CEILZ(*expression*)

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42.](#)

Comparisons

Unlike the CEIL function, the CEILZ function uses zero fuzzing. If the argument is within 1E-12 of an integer, the CEIL function fuzzes the result to be equal to that integer. The CEILZ function does not fuzz the result. Therefore, with the CEILZ function, you might get unexpected results.

Example

The following statements illustrate the CEILZ function:

Statements	Results
<code>select ceilz(2.1);</code>	3
<code>select ceilz(3);</code>	3
<code>select ceilz(1+1.e-11);</code>	2
<code>select ceilz(223.456);</code>	224
<code>select ceilz(-223.456);</code>	-223

See Also

Functions:

- [“CEIL Function” on page 212](#)
- [“FLOOR Function” on page 235](#)
- [“FLOORZ Function” on page 236](#)

CHARACTER_LENGTH Function

Returns the number of characters in a string of any data type.

Category: Character

Alias: CHAR_LENGTH

Returned data type: BIGINT

Note: The CHARACTER_LENGTH function counts trailing blanks. If you do not want to count trailing blanks, then you must use the [“TRIM Function” on page 313](#) to remove them.

Syntax

CHARACTER_LENGTH(*expression*)

Arguments

expression

specifies any valid expression.

Data type All data types are valid.

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Comparisons

The CHARACTER_LENGTH function returns the number of characters in a character string, but counts a multibyte character as a single character. The OCTET_LENGTH function counts a multibyte character as multiple characters.

Example

The following statements illustrate the CHARACTER_LENGTH function:

Statements	Results
<code>select character_length('December');</code>	8
<code>select character_length('FedSQL ');</code>	7

See Also

Functions:

- [“OCTET_LENGTH Function” on page 277](#)

COALESCE Function

Returns the first non-null or nonmissing value from a list of numeric arguments.

Category:	Mathematical
Returned data type:	DOUBLE

Syntax

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

Arguments

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

Data type	DOUBLE
See	“<sql-expression>” on page 339 “FedSQL Expressions” on page 42.

Details

COALESCE accepts one or more numeric expressions. The COALESCE function checks the value of each expression in the order in which they are listed and returns the first non-null or nonmissing value. If only one value is listed, then the COALESCE function returns the value of that argument. If all the values of all expressions are null or missing, then the COALESCE function returns a null or a missing value depending on whether you are in ANSI mode or SAS mode. For more information, see [“How FedSQL Processes Nulls and SAS Missing Values” on page 18.](#)

Comparisons

The COALESCE function searches numeric expressions, whereas the COALESCEC function searches character expressions.

Example

The following statement illustrates the COALESCE function:

Statements	Results
<pre>select COALESCE(., .A, 33, 22, 44, .);</pre>	33

See Also

Functions:

- [“COALESCEC Function” on page 217](#)

COALESCEC Function

Returns the first non-null or nonmissing value from a list of character arguments.

Category: Character

Returned data type: NCHAR

Syntax

COALESCEC(*expression* [, ...*expression*])

Arguments

expression

specifies any valid expression that evaluates to a character value.

Data type NCHAR

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42.](#)

Details

COALESCEC accepts one or more character expressions. The COALESCEC function checks the value of each expression in the order in which they are listed and returns the first non-null or nonmissing value. If only one value is listed, then the COALESCEC function returns the value of that expression. If all the values of all expressions are null or missing, then the COALESCEC function returns a null or missing value depending on whether you are in ANSI mode or SAS mode. For more information, see [“How FedSQL Processes Nulls and SAS Missing Values” on page 18.](#)

Comparisons

The COALESCEC function searches character expressions, whereas the COALESCE function searches numeric expressions.

Example

The following statements illustrate the COALESCEC function:

Statements	Results
<pre>select coalescec('', 'Hello');</pre>	Hello
<pre>select coalescec('', 'Goodbye', 'Hello');</pre>	Goodbye

See Also

Functions:

- [“COALESCE Function” on page 216](#)

COS Function

Returns the cosine in radians.

Category: Trigonometric

Returned data type: DOUBLE

Syntax

`COS(expression)`

Arguments

expression

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

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Example

The following statements illustrate the COS function:

Statements	Results
<code>select cos(0.5);</code>	0.877583
<code>select cos(0);</code>	1
<code>select cos(3.14159/3);</code>	0.50003

See Also

Functions:

- [“ACOS Function” on page 191](#)
- [“COSH Function” on page 219](#)
- [“SIN Function” on page 297](#)
- [“TAN Function” on page 309](#)

COSH Function

Returns the hyperbolic cosine in radians.

Category: Trigonometric

Returned data type: DOUBLE

Syntax

COSH(*expression*)

Arguments

expression

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

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The COSH function returns the hyperbolic cosine of the argument, given by the following equation.

$$e^{argument} + e^{-argument} / 2$$

Example

The following statements illustrate the COSH function:

Statements	Results
<code>select cosh(0);</code>	1
<code>select cosh(-5.0);</code>	74.20995
<code>select cosh(4.37);</code>	39.52814
<code>select cosh(0.5);</code>	1.127626

See Also

Functions:

- [“ACOS Function” on page 191](#)
- [“COS Function” on page 218](#)

- [“SINH Function” on page 298](#)
- [“TANH Function” on page 310](#)

COT Function

Returns the tangent in radians.

Category: Trigonometric

Returned data type: DOUBLE

Syntax

COT(*expression*)

Arguments

expression

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

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Example

The following statements illustrate the COT function:

Statements	Results
<code>select cot(0);</code>	.
<code>select cot(0.5)</code>	1.830488
<code>select cot(5.79);</code>	-1.86051
<code>select cot(1);</code>	0.642093

See Also

Functions:

- [“ATAN Function” on page 193](#)
- [“COS Function” on page 218](#)
- [“TAN Function” on page 309](#)

COUNT Function

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

Category: Aggregate

Alias: N

Returned data type: BIGINT

Syntax

Form 1: **COUNT**(*expression*)

Form 2: **COUNT**(*)

Form 3: **COUNT**([**DISTINCT**] *expression*)

Arguments

expression

specifies any valid SQL expression.

Data type All data types are valid.

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

*

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

DISTINCT

returns the number of unique values, excluding null values.

Details

You use the COUNT function in a SELECT statement to return the requested number of rows in a table.

The following list describes what is returned by using the different versions of the COUNT function:

Form 1: **COUNT**(*expression*)

returns the number of rows from a table that do not have a null value.

Form 2: **COUNT**(*)

returns the number of rows in a table.

Form 3: **COUNT**(**DISTINCT** *expression*)

returns the number of rows in *expression* that have unique values. SAS missing values are included in the results. Null values are not included in the results.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in

a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Example

Table: [WORLDTEMPS on page 542](#)

The following statements illustrate the COUNT function:

Statements	Results
select count(AvgHigh) from worldtemps;	11
select count(*) from worldtemps;	12
select count(distinct AvgHigh) from worldtemps;	8

See Also

SELECT Statement Clauses:

- [“SELECT Clause” on page 389](#)
- [“GROUP BY Clause” on page 400](#)
- [“HAVING Clause” on page 401](#)

CSS Function

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

Categories:	Aggregate Descriptive Statistics
Returned data type:	DOUBLE

Syntax

CSS(*expression*)

Arguments

<i>expression</i>	specifies any valid SQL expression.
Data type	DOUBLE, FLOAT, REAL
See	“<sql-expression>” on page 339 “FedSQL Expressions” on page 42

Details

The corrected sum of squares is the sum of squared deviations (differences) from the mean.

Null values and SAS missing values are ignored and are not included in the computation.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Comparisons

The USS function returns the uncorrected sum of squares. The CSS function returns the corrected sum of squares of all values.

Example

Table: [DENSITIES on page 539](#)

The following statement illustrates the CSS function:

Statements	Results
<pre>select css(density) from densities;</pre>	211483.1

See Also

Functions:

- [“USS Function” on page 315](#)

SELECT Statement Clauses:

- [“SELECT Clause” on page 389](#)
- [“GROUP BY Clause” on page 400](#)
- [“HAVING Clause” on page 401](#)

CURRENT_DATE Function

Returns the current date for the time zone.

Category: Date and Time

Returned data type: DATE

Syntax

CURRENT_DATE

Comparisons

The **CURRENT_DATE** function returns the current date for the timezone. The **CURRENT_TIMESTAMP_GMT()** function returns the current GMT date.

Example

The following statement illustrates the **CURRENT_DATE** function:

Statements	Results
<code>select current_date;</code>	05FEB2014

See Also

Functions:

- [“CURRENT_TIME Function” on page 225](#)
- [“CURRENT_TIME_GMT Function” on page 225](#)
- [“CURRENT_TIMESTAMP Function” on page 226](#)
- [“CURRENT_TIMESTAMP_GMT Function” on page 227](#)

CURRENT_LOCALE Function

Returns the five character name of the current locale.

Category: Character

Returned data type: CHAR(5)

Syntax

CURRENT_LOCALE()

Example

The following statement illustrates the **CURRENT_LOCALE** function:

Statements	Results
<code>select current_locale();</code>	en_US

CURRENT_TIME Function

Returns the current time for your time zone.

Category: Date and Time

Alias: LOCALTIME

Returned data type: TIME

Syntax

CURRENT_TIME

Comparisons

The **CURRENT_TIME** function returns the current time for your time zone. The **CURRENT_TIME_GMT** function returns the current time in GMT.

Example

The following statement illustrates the **CURRENT_TIME** function:

Statements	Results
<code>select current_time;</code>	11:00:49
<code>select localtime;</code>	11:00:49

See Also

Functions:

- [“CURRENT_DATE Function” on page 223](#)
- [“CURRENT_TIME_GMT Function” on page 225](#)
- [“CURRENT_TIMESTAMP Function” on page 226](#)
- [“CURRENT_TIMESTAMP_GMT Function” on page 227](#)

CURRENT_TIME_GMT Function

Returns the current GMT time.

Category: Date and Time

Returned data type: TIME

Syntax

CURRENT_TIME_GMT()

Comparisons

The **CURRENT_TIME_GMT** function returns the current GMT time. The **CURRENT_TIME** function returns the current time for your time zone.

Example

The following statement illustrates the **CURRENT_TIME_GMT** function:

Statements	Results
<code>select current_time_gmt();</code>	15:00:49

See Also

Functions:

- [“CURRENT_DATE Function” on page 223](#)
- [“CURRENT_TIME Function” on page 225](#)
- [“CURRENT_TIMESTAMP Function” on page 226](#)
- [“CURRENT_TIMESTAMP_GMT Function” on page 227](#)

CURRENT_TIMESTAMP Function

Returns the date and time for your time zone.

Category: Date and Time

Alias: LOCALTIMESTAMP

Returned data type: TIMESTAMP

Syntax

CURRENT_TIMESTAMP

Comparisons

The **CURRENT_TIMESTAMP** function returns the date and time for your time zone.

Example

The following statement illustrates the **CURRENT_TIMESTAMP** function:

Statements	Results
<code>select current_timestamp;</code>	05FEB2014:11:06:05
<code>select localtimestamp;</code>	05FEB2014:11:06:05

See Also

Functions:

- [“CURRENT_DATE Function” on page 223](#)
- [“CURRENT_TIME Function” on page 225](#)
- [“CURRENT_TIME_GMT Function” on page 225](#)
- [“CURRENT_TIMESTAMP_GMT Function” on page 227](#)

CURRENT_TIMESTAMP_GMT Function

Returns the current GMT date and time.

Category: Date and Time

Returned data type: TIMESTAMP

Syntax

`CURRENT_TIMESTAMP_GMT()`

Comparisons

The `CURRENT_TIMESTAMP_GMT` function returns the current GMT date and time. The `CURRENT_TIMESTAMP` function returns the current date and time for your time zone.

Example

The following statement illustrates the `CURRENT_TIMESTAMP_GMT` function:

Statements	Results
<code>select current_timestamp_gmt();</code>	05FEB2014:17:06:34

See Also

Functions:

- [“CURRENT_DATE Function” on page 223](#)
- [“CURRENT_TIME Function” on page 225](#)

- “CURRENT_TIMESTAMP Function” on page 226
- “CURRENT_TIME_GMT Function” on page 225

CV Function

Returns the coefficient of variation.

Category: Descriptive Statistics

Returned data type: DOUBLE

Syntax

CV(*expression-1*, *expression-2* [, ...*expression-n*])

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Requirement At least two arguments are required.

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Example

The following statements illustrate the CV function:

Statements	Results
<code>select cv(5,9,3,6);</code>	43.4782608695652
<code>select cv(5,8,9,6,.);</code>	26.0820265478651
<code>select cv(8,9,6,.);</code>	19.9242421519819

DATE Function

Returns the current date as a SAS date value.

Category: Date and Time

Alias: TODAY

Returned data type: DOUBLE

Syntax

DATE()

Comparisons

The DATE function does not take any arguments. The SAS date value returned is the number of days from January 1, 1960 to the current date.

For more information about how FedSQL handles dates, see [“Dates and Times in FedSQL” on page 51](#).

Example

The following statement illustrates the DATE function:

Statements	Results
<pre>select date();</pre>	19611

See Also

Functions:

- [“TODAY Function” on page 312](#)

DATEJUL Function

Converts a Julian date to a SAS date value.

Category: Date and Time

Returned data type: DOUBLE

Syntax

DATEJUL(*julian-date*)

Arguments

julian-date

specifies any valid expression that evaluates to a numeric value and that represents a Julian date. A Julian date is a date in the form *yyddd* or *yyyyddd*, where *yy* or *yyyy* is a two-digit or four-digit integer that represents the year and *ddd* is the number of the day of the year. The value of *ddd* must be between 1 and 365 (or 366 for a leap year).

Data type DOUBLE

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

Details

A SAS date value is the number of days from January 1, 1960 to a specified date. The DATEJUL function returns the number of days from January 1, 1960 to the Julian date specified in *julian-date*.

For more information about how dates are handled in FedSQL, see [“Dates and Times in FedSQL” on page 51](#).

Example

The following statements illustrate the DATEJUL function:

Statements	Results
<pre>select datejul(11365);</pre>	18992

See Also

Functions:

- [“JULDATE Function” on page 248](#)

DATEPART Function

Returns the date as year, month, and day.

Category: Date and Time

Returned data type: DATE

Syntax

DATEPART(*ts*)

Arguments

ts
specifies the timestamp.

Example

The following statement illustrates the DATEPART function:

Statements	Results
<pre>select datepart(timestamp '2013-09-24 14:46:58');</pre>	24SEP2013

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

See Also

Functions:

- [“MAKEDATE Function” on page 260](#)
- [“MAKETIME Function” on page 260](#)
- [“MAKETIMESTAMP Function” on page 261](#)
- [“TIMEPART Function” on page 311](#)

DAY Function

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

Category: Date and Time

Returned data type: TINYINT

Syntax

DAY(*date* | *datetime*)

Arguments

date

specifies any valid expression that represents a date value.

Data type DATE

See [“FedSQL Expressions” on page 42](#)

datetime

specifies any valid expression that represents a datetime value.

Data type TIMESTAMP

See [“FedSQL Expressions” on page 42](#)

Example

Table: [CUSTONLINE on page 538](#)

The following statement illustrates the DAY function:

Statements	Results
select day(endtime) from custonline;	1
	2
	15
	1
	1
	2
	16
	1
	1
	15
select day(current_time);	17

See Also

- [“Dates and Times in FedSQL” on page 51](#)

Functions:

- [“HOUR Function” on page 245](#)
- [“MINUTE Function” on page 272](#)
- [“MONTH Function” on page 275](#)
- [“SECOND Function” on page 295](#)
- [“YEAR Function” on page 318](#)

DEGREES Function

Returns the number of degrees for an angle in radians.

Category:	Trigonometric
Returned data type:	DOUBLE

Syntax

DEGREES(*expression*)

Arguments

expression
specifies any valid expression that evaluates to an angle specified in radians.

Data type BIGINT, DOUBLE, FLOAT, INTEGER, REAL, SMALLINT, TINYINT.

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

If *expression* is a null value, then the DEGREES function returns a null value. If the result is a number that does not fit into the range of a DOUBLE data type, the DEGREES function fails.

Example

The following statements illustrate the DEGREES function:

Statements	Results
<code>select degrees(2*pi());</code>	360
<code>select degrees(pi());</code>	180
<code>select degrees(pi()/2);</code>	90
<code>select degrees(pi()/4);</code>	45

See Also

Functions:

- [“RADIANS Function” on page 290](#)

E Function

Returns the natural logarithm, e.

Category: Mathematical

Returned data type: DOUBLE

Syntax

`E()`

Details

The E function returns the value of the natural logarithm, e, 2.7182818.

Comparisons

The E function takes no argument and returns the value 2.7182818. The EXP function takes an argument and raises e to the power that is supplied by the argument.

Example

The following statement illustrates the E function:

Statements	Results
<code>select e();</code>	2.718282

See Also

Functions:

- [“EXP Function” on page 234](#)
- [“LOG Function” on page 254](#)

EXP Function

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

Category: Mathematical

Returned data type: DOUBLE

Syntax

EXP(*expression*)

Arguments

expression

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

Data type BIGINT, DOUBLE, FLOAT, INTEGER, REAL, SMALLINT, TINYINT

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The EXP function raises the constant e , which is approximately given by 2.71828, to the power that is supplied by the argument. The result is limited by the maximum value of a double decimal value on the computer.

Comparisons

The EXP function takes an argument and raises e to the power that is supplied by the argument. The E function takes no argument and returns the value 2.7182818.

Example

The following statements illustrate the EXP function:

Statements	Results
<code>select exp(1.0);</code>	2.718282
<code>select exp(0);</code>	1

See Also

Functions:

- [“E Function” on page 233](#)
- [“LOG Function” on page 254](#)

FLOOR Function

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

Category: Truncation

Returned data type: DECIMAL, DOUBLE, NUMERIC

Syntax

FLOOR(*expression*)

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Data type DECIMAL, DOUBLE, NUMERIC

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

If *expression* is within 1E-12 of an integer, the function returns that integer. If the result is a number that does not fit into the range of a DOUBLE, the FLOOR function fails.

If the argument is DECIMAL, the result is DECIMAL. Otherwise, the argument is converted to DOUBLE (if not so already), and the result is DOUBLE.

Comparisons

The FLOOR function fuzzes the results so that if the results are within 1E-12 of an integer, the FLOOR function returns that integer. The FLOORZ function uses zero fuzzing. Therefore, with the FLOORZ function, you might get unexpected results.

Example

The following statement illustrates the FLOOR function:

Statements	Results
<code>select floor(1.95);</code>	1
<code>select floor(density) from densities;</code>	67 306 30 323 20 383 31 309 6 247

See Also

Functions:

- [“CEIL Function” on page 212](#)
- [“CEILZ Function” on page 213](#)
- [“FLOORZ Function” on page 236](#)

FLOORZ Function

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

Category:	Truncation
Returned data type:	DOUBLE

Syntax

FLOORZ(*expression*)

Arguments

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

Data type	DOUBLE
See	“<sql-expression>” on page 339 “FedSQL Expressions” on page 42

Comparisons

Unlike the FLOOR function, the FLOORZ function uses zero fuzzing. If the argument is within 1E-12 of an integer, the FLOOR function fuzzes the result to be equal to that integer. The FLOORZ function does not fuzz the result. Therefore, with the FLOORZ function, you might get unexpected results.

Example

The following statements illustrate the FLOORZ function:

Statements	Results
<code>select floorz(-2.4);</code>	-3
<code>select floorz(-1.6);</code>	-2
<code>select floorz(density) from densities;</code>	67 306 30 323 20 383 31 309 6 247

See Also

Functions:

- [“CEIL Function” on page 212](#)
- [“CEILZ Function” on page 213](#)
- [“FLOOR Function” on page 235](#)

GAMMA Function

Returns the value of the gamma function.

Category: Mathematical

Returned data type: DOUBLE

Syntax

GAMMA(*expression*)

Arguments

<i>expression</i>	specifies any valid expression that evaluates to a numeric value.
Restriction	Nonpositive integers are invalid.
Data type	DOUBLE
See	“<sql-expression>” on page 339
	“FedSQL Expressions” on page 42

Details

The GAMMA function returns the integral, which is given by the following equation.

$$GAMMA(x) = \int_0^{\infty} t^{x-1} e^{-t} dt.$$

For positive integers, GAMMA(x) is (x – 1)!. This function is commonly denoted by $\Gamma(x)$.

Example

The following statement illustrates the GAMMA function:

Statements	Results
<code>select gamma(6);</code>	120

GCD Function

Returns the greatest common divisor for a set of integers.

Category:	Mathematical
Returned data type:	DOUBLE

Syntax

`GCD(expression-1, expression-2 [, ...expression-n])`

Arguments

<i>expression</i>	specifies any valid expression that evaluates to a numeric value.
Requirement	At least two arguments are required.

Data type	DOUBLE
See	“<sql-expression>” on page 339
	“FedSQL Expressions” on page 42

Details

The GCD (greatest common divisor) function returns the greatest common divisor of one or more integers. For example, the greatest common divisor for 30 and 42 is 6. The greatest common divisor is also called the highest common factor.

Example

The following statements illustrate the GCD function:

Statements	Results
<code>select gcd(5,15);</code>	5
<code>select gcd(36,45);</code>	9

See Also

Functions:

- [“LCM Function” on page 253](#)

GEOMEAN Function

Returns the geometric mean.

Category:	Descriptive Statistics
Returned data type:	DOUBLE

Syntax

GEOMEAN(*expression* [, ...*expression*])

Arguments

expression

is any valid expression that evaluates to a nonnegative numeric value.

Data type	DOUBLE
See	“<sql-expression>” on page 339
	“FedSQL Expressions” on page 42

Details

If any argument is negative, then the result is a null or missing value. A message appears in the log that the negative argument is invalid. If any argument is zero, then the geometric mean is zero. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the geometric mean of the non-null or nonmissing values.

Let n be the number of arguments with non-null or nonmissing values, and let x_1, x_2, \dots, x_n be the values of those arguments. The geometric mean is the n^{th} root of the product of the values:

$$\sqrt[n]{(x_1 * x_2 * \dots * x_n)}$$

Equivalently, the geometric mean is shown in this equation.

$$\exp\left(\frac{(\log(x_1) + \log(x_2) + \dots + \log(x_n))}{n}\right)$$

Floating-point arithmetic often produces tiny numerical errors. Some computations that result in zero when exact arithmetic is used might result in a tiny nonzero value when floating-point arithmetic is used. Therefore, GEOMEAN fuzzes the values of arguments that are approximately zero. When the value of one argument is extremely small relative to the largest argument, the former argument is treated as zero. If you do not want SAS to fuzz the extremely small values, then use the GEOMEANZ function.

Comparisons

The MEAN function returns the arithmetic mean (average), and the HARMEAN function returns the harmonic mean, whereas the GEOMEAN function returns the geometric mean of the non-null or nonmissing values. Unlike GEOMEANZ, GEOMEAN fuzzes the values of the arguments that are approximately zero.

Example

The following statements illustrate the GEOMEAN function:

Statements	Results
<code>select geomean(1,2,2,4);</code>	2
<code>select geomean(.,2,4,8);</code>	4

See Also

Functions:

- [“GEOMEANZ Function” on page 241](#)
- [“HARMEAN Function” on page 242](#)
- [“HARMEANZ Function” on page 243](#)
- [“MEAN Function” on page 269](#)

GEOMEANZ Function

Returns the geometric mean, using zero fuzzing.

Category: Descriptive Statistics

Returned data type: DOUBLE

Syntax

GEOMEANZ(*expression* [, ...*expression*])

Arguments

expression

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

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

If any argument is negative, then the result is a null or missing value. A message appears in the log that the negative argument is invalid. If any argument is zero, then the geometric mean is zero. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the geometric mean of the non-null or nonmissing values.

Let n be the number of arguments with non-null or nonmissing values, and let x_1, x_2, \dots, x_n be the values of those arguments. The geometric mean is the n^{th} root of the product of the values:

$$\sqrt[n]{(x_1 * x_2 * \dots * x_n)}$$

Equivalently, the geometric mean is shown in this equation.

$$\exp \left(\frac{(\log(x_1) + \log(x_2) + \dots + \log(x_n))}{n} \right)$$

Comparisons

The MEAN function returns the arithmetic mean (average), and the HARMEAN function returns the harmonic mean, whereas the GEOMEANZ function returns the geometric mean of the non-null or nonmissing values. Unlike GEOMEAN, GEOMEANZ does not fuzz the values of the arguments that are approximately zero.

Example

The following statements illustrate the GEOMEANZ function:

Statements	Results
<code>select geomeanz(1,2,2,4);</code>	2
<code>select geomeanz(.,2,4,8);</code>	4

See Also

Functions:

- [“GEOMEAN Function” on page 239](#)
- [“HARMEAN Function” on page 242](#)
- [“HARMEANZ Function” on page 243](#)
- [“MEAN Function” on page 269](#)

HARMEAN Function

Returns the harmonic mean.

Category: Descriptive Statistics

Returned data type: DOUBLE

Syntax

HARMEAN(*expression* [, ...*expression*])

Arguments

expression

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

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

If any argument is negative, then the result is a null or missing value. A message appears in the log that the negative argument is invalid. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the harmonic mean of the non-null or nonmissing values.

If any argument is zero, then the harmonic mean is zero. Otherwise, the harmonic mean is the reciprocal of the arithmetic mean of the reciprocals of the values.

Let n be the number of arguments with non-null or nonmissing values, and let x_1, x_2, \dots, x_n be the values of those arguments. The harmonic mean is shown in this equation.

$$\frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \dots + \frac{1}{x_n}}$$

Floating-point arithmetic often produces tiny numerical errors. Some computations that result in zero when exact arithmetic is used might result in a tiny nonzero value when floating-point arithmetic is used. Therefore, HARMEAN fuzzes the values of arguments that are approximately zero. When the value of one argument is extremely small relative to the largest argument, the former argument is treated as zero. If you do not want SAS to fuzz the extremely small values, then use the HARMEANZ function.

Comparisons

The MEAN function returns the arithmetic mean (average), and the GEOMEAN function returns the geometric mean, whereas the HARMEAN function returns the harmonic mean of the non-null or nonmissing values. Unlike HARMEANZ, HARMEAN fuzzes the values of the arguments that are approximately zero.

Example

The following statements illustrate the HARMEAN function:

Statements	Results
<code>select harmean(1,2,4,4);</code>	2
<code>select harmean(.,4,12,24);</code>	8

See Also

Functions:

- [“GEOMEAN Function” on page 239](#)
- [“GEOMEANZ Function” on page 241](#)
- [“HARMEANZ Function” on page 243](#)
- [“MEAN Function” on page 269](#)

HARMEANZ Function

Returns the harmonic mean, using zero fuzzing.

Category: Descriptive Statistics

Returned data type: DOUBLE

Syntax

HARMEANZ(*expression* [, ...*expression*])

Arguments

expression

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

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

If any argument is negative, then the result is a null or value. A message appears in the log that the negative argument is invalid. If all the arguments are null or values, then the result is a null or value. Otherwise, the result is the harmonic mean of the non-null or nonmissing values.

If any argument is zero, then the harmonic mean is zero. Otherwise, the harmonic mean is the reciprocal of the arithmetic mean of the reciprocals of the values.

Let n be the number of arguments with non-null or nonmissing values, and let x_1, x_2, \dots, x_n be the values of those arguments. The harmonic mean is shown in this equation.

$$\frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \dots + \frac{1}{x_n}}$$

Comparisons

The MEAN function returns the arithmetic mean (average), and the GEOMEAN function returns the geometric mean, whereas the HARMEANZ function returns the harmonic mean of the non-null or nonmissing values. Unlike HARMEAN, HARMEANZ does not fuzz the values of the arguments that are approximately zero.

Example

The following statements illustrate the HARMEANZ function:

Statements	Results
<code>select harmeanz(1,2,4,4);</code>	2
<code>select harmeanz(.,4,12,24);</code>	8

See Also

Functions:

- [“GEOMEAN Function” on page 239](#)

- [“GEOMEANZ Function” on page 241](#)
- [“HARMEAN Function” on page 242](#)
- [“MEAN Function” on page 269](#)

HOUR Function

Returns the hour from a time or datetime value.

Category: Date and Time

Returned data type: TINYINT

Syntax

HOUR(*time* | *datetime*)

Arguments

time

specifies any valid expression that represents a time value.

Data type TIME

See [“FedSQL Expressions” on page 42](#)

datetime

specifies any valid expression that represents a datetime value.

Data type TIMESTAMP

See [“FedSQL Expressions” on page 42](#)

Details

In Hive, when the HOUR function is used to search for a specific value, the function will fail if there is more than one column of type TIME in the table.

Example

Table: [CUSTONLINE on page 538](#)

The following statement illustrates the HOUR function:

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

See Also

- [“Dates and Times in FedSQL” on page 51](#)

Functions:

- [“DAY Function” on page 231](#)
- [“MINUTE Function” on page 272](#)
- [“MONTH Function” on page 275](#)
- [“SECOND Function” on page 295](#)

IFNULL Function

Checks the value of the first expression and, if it is null or a SAS missing value, returns the second expression.

Category:	Special
Returned data type:	Any data type, depending on the context

Syntax

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

Arguments

<i>expression</i>	specifies any valid expression.
Data type	All data types are valid.
See	“<sql-expression>” on page 339
	“FedSQL Expressions” on page 42

Details

The IFNULL function returns the first expression if it is not null. You can use the IFNULL function to replace a null or SAS missing value value with another value.

Comparisons

The NULLIF expression compares two expressions and returns a null value if the two expressions are equal. The IFNULL function checks the value of the first expression and if it is null, returns the second expression; otherwise, returns the first expression.

Example

Table: [WORLDCITYCOORDS](#) on page 542

In the table WORLDTEMPS, the city for China in the last row contains a missing value.

Statements	Results
select ifnull(AvgHigh, 0) as "AvgHigh: 0 indicates null/missing" from worldtemps;	AvgHigh: 0 indicates null/missing 90 70 86 90 97 83 76 89 90 89 0 78

See Also

Expressions:

- [“NULLIF Expression”](#) on page 338

IQR Function

Returns the interquartile range.

Category: Descriptive Statistics

Returned data type: DOUBLE

Syntax

IQR(*expression* [, ...*expression*])

Arguments

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

Data type	DOUBLE
See	“<sql-expression>” on page 339
	“FedSQL Expressions” on page 42

Details

If all arguments have null or missing values, the result is a null or missing value depending on whether you are in ANSI mode or SAS mode. For more information, see [“How FedSQL Processes Nulls and SAS Missing Values” on page 18](#).

Otherwise, the result is the interquartile range of the non-null or nonmissing values. The formula for the interquartile range is the same as the one that is used in the Base SAS UNIVARIATE procedure. For more information, see *Base SAS Procedures Guide: Statistical Procedures*.

Example

The following statement illustrates the IQR function:

Statements	Results
<pre>select iqr(2,4,1,3,999999);</pre>	2

See Also

- Functions:
- [“MAD Function” on page 259](#)
 - [“PCTL Function” on page 279](#)

JULDATE Function

Returns the Julian date from a SAS date value.

Category:	Date and Time
Returned data type:	DOUBLE

Syntax

JULDATE(*date*)

Arguments

date

specifies any valid expression that represents a SAS date value.

Data type DOUBLE

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

Details

A SAS date value is a number that represents the number of days from January 1, 1960 to a specific date. The JULDATE function converts a SAS date value to a Julian date. If *date* falls within the 100-year span defined by the system option YEARCUTOFF=, the result has three, four or five digits: In a five-digit result, the first two digits represent the year, and the next three digits represent the day of the year (1 to 365, or 1 to 366 for leap years). As leading zeros are dropped from the result, the year portion of a Julian date can be omitted (for years ending in 00) or it can have only one digit (for years ending 01–09). Otherwise, the result has seven digits: the first four digits represent the year, and the next three digits represent the day of the year.

For years that end between 00–09, you can format the five-digit Julian date by using the Z5. format.

For more information about how FedSQL handles dates, see [“Dates and Times in FedSQL” on page 51](#).

Comparisons

The function JULDATE7 is similar to JULDATE except that JULDATE7 always returns a four-digit year. Thus, JULDATE7 is year 2000 compliant because it eliminates the need to consider the implications of a two-digit year.

Example

The following statements illustrate the JULDATE function:

Statements	Results
<code>select juldate(mdy(12,31,2013));</code>	7365
<code>select put(juldate(mdy(12,31,2013)),z5.);</code>	07365
<code>select juldate(mdy(9,1,1999));</code>	99244
<code>select juldate(mdy(7,1,1886));</code>	1886182

See Also

Functions:

- [“DATEJUL Function” on page 229](#)
- [“JULDATE7 Function” on page 250](#)

JULDATE7 Function

Returns a seven-digit Julian date from a SAS date value.

Category: Date and Time

Returned data type: Return value data: DOUBLE

Syntax

JULDATE7(*date*)

Arguments

date

specifies any valid expression that represents a SAS date value.

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

A SAS date value is a number that represents the number of days from January 1, 1960 to a specific date. The JULDATE7 function returns a seven-digit Julian date from a SAS date value. The first four digits represent the year, and the next three digits represent the day of the year.

For more information about how FedSQL handles dates, see [“Dates and Times in FedSQL” on page 51](#).

Comparisons

The function JULDATE7 is similar to JULDATE except that JULDATE7 always returns a four-digit year. Thus, JULDATE7 is year 2000 compliant because it eliminates the need to consider the implications of a two-digit year.

Example

The following statements illustrate the JULDATE7 function:

Statements	Results
<pre>select juldate7(mdy(12,31,2006));</pre>	2007365
<pre>select juldate7(mdy(12,31,2016));</pre>	2016366

See Also

Functions:

- [“JULDATE Function” on page 248](#)

LARGEST Function

Returns the k th largest non-null or nonmissing value.

Category: Descriptive Statistics

Returned data type: DOUBLE

Syntax

LARGEST(k , *expression* [, ...*expression*])

Arguments

k

specifies any valid expression that evaluates to a numeric value that represents the largest value to return. For example, if k is 2, the LARGEST function returns the second largest value from the list of expressions.

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

expression

specifies any valid expression that evaluates to a numeric value and that is to be searched.

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

If k is null or missing, less than zero, or greater than the number of values, the result is a null or missing value. Otherwise, if k is greater than the number of non-null or nonmissing values, the result is a null or missing value.

Example

The following statements illustrate the LARGEST function:

Statements	Results
<code>select largest(1, 456, 789, .Q, 123);</code>	789
<code>select largest(2, 456, 789, .Q, 123);</code>	456
<code>select largest(3, 456, 789, .Q, 123);</code>	123
<code>select largest(4, 456, 789, .Q, 123);</code>	.

See Also

Functions:

- [“ORDINAL Function” on page 278](#)
- [“PCTL Function” on page 279](#)
- [“SMALLEST Function” on page 300](#)

KURTOSIS Function

Returns the kurtosis of all values in an expression.

Categories: Aggregate
Descriptive Statistics

Returned data type: DOUBLE

Syntax

KURTOSIS(*expression*)

Arguments

expression
specifies any valid SQL expression.

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

Kurtosis is primarily a measure of the heaviness of the tails of a distribution. Large values of kurtosis indicate that the distribution has heavy tails.

Null values and SAS missing values are ignored and are not included in the computation.

At least four nonnull or nonmissing arguments are required. Otherwise, the function returns a null value. If all nonnull or nonmissing arguments have equal values, the kurtosis is mathematically undefined and the KURTOSIS function returns a null value.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Example

Table: [WORLDTEMPS on page 542](#)

The following statement illustrates the KURTOSIS function:

Statements	Results
<code>select kurtosis(AvgLow) from worldtemps;</code>	-0.87431

See Also

Functions:

- [“STDDEV Function” on page 303](#)

SELECT Statement Clauses:

- [“SELECT Clause” on page 389](#)
- [“GROUP BY Clause” on page 400](#)
- [“HAVING Clause” on page 401](#)

LCM Function

Returns the least common multiple for a set of integers.

Category: Mathematical

Returned data type: DOUBLE

Syntax

`LCM(expression-1, expression-2 [, ...expression-n])`

Arguments

expression

specifies any valid expression that evaluates to an integer.

Requirement	At least two arguments are required.
Data type	DOUBLE
See	“<sql-expression>” on page 339
	“FedSQL Expressions” on page 42

Details

The least common multiple is the smallest number that two or more numbers will divide into evenly.

Example

The following statements illustrate the LCM function:

Statements	Results
<code>select lcm(1,5,3,0);</code>	.
<code>select lcm(25,70,85,130);</code>	77350
<code>select lcm(33,78);</code>	858

See Also

Functions:

- [“GCD Function” on page 238](#)

LOG Function

Returns the natural logarithm (base e) of a numeric value expression.

Category:	Mathematical
Returned data type:	DOUBLE

Syntax

LOG(*expression*)

Arguments

expression

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

Data type	DOUBLE
-----------	--------

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Example

The following statements illustrate the LOG function:

Statements	Results
<code>select log(1.0);</code>	0
<code>select log(10.0);</code>	2.302585

See Also

Functions:

- [“E Function” on page 233](#)
- [“LOG10 Function” on page 256](#)
- [“LOG2 Function” on page 255](#)

LOG2 Function

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

Category: Mathematical

Returned data type: DOUBLE

Syntax

LOG2(*expression*)

Arguments

expression

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

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Example

The following statements illustrate the LOG2 function:

Statements	Results
<code>select log2(8.0);</code>	3
<code>select log2(4);</code>	2

See Also

Functions:

- [“LOG Function” on page 254](#)
- [“LOG10 Function” on page 256](#)

LOG10 Function

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

Category: Mathematical

Returned data type: DOUBLE

Syntax

LOG10(*expression*)

Arguments

expression

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

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Example

The following statements illustrate the LOG10 function:

Statements	Results
<code>select log10(1.0);</code>	0
<code>select log10(10.0);</code>	1
<code>select log10(100.0);</code>	2

See Also

Functions:

- [“LOG Function” on page 254](#)
- [“LOG2 Function” on page 255](#)

LOGBETA Function

Returns the logarithm of the beta function.

Category: Mathematical

Returned data type: DOUBLE

Syntax

LOGBETA(*a*, *b*)

Arguments

a

is the first shape parameter, where $a > 0$.

Data type DOUBLE

b

is the second shape parameter, where $b > 0$.

Data type DOUBLE

Details

The LOGBETA function is mathematically given by the equation

$$\log(\beta(a, b)) = \log(\Gamma(a)) + \log(\Gamma(b)) - \log(\Gamma(a + b))$$

In the equation, $\Gamma(\cdot)$ is the gamma function.

If the expression cannot be computed, LOGBETA returns a missing value.

Example

The following statement illustrates the LOGBETA function:

Statement	Result
<pre>select logbeta(5,3);</pre>	-4.6539603501575

See Also

Functions:

- [“BETA Function” on page 198](#)

LOWCASE Function

Converts all letters in a character expression to lowercase.

Category:	Character
Alias:	LOWER
Returned data type:	VARCHAR, NVARCHAR

Syntax

LOWCASE(*expression*)

Arguments

expression

specifies any valid expression that evaluates to a character string.

Requirement	Literal character expressions must be enclosed in single quotation marks.
--------------------	---

Data type	CHAR, NCHAR
------------------	-------------

See	“<sql-expression>” on page 339
	“FedSQL Expressions” on page 42

Details

The LOWCASE function copies a character expression, converts all uppercase letters to lowercase letters, and returns the altered value as a result.

Comparisons

The UPCASE function converts all letters in an argument to uppercase letters. The LOWCASE function converts all letters in an argument to lowercase letters.

Example

The following statement illustrates the LOWCASE function:

Statements	Results
<pre>select lowercase('INTRODUCTION');</pre>	<pre>introduction</pre>

See Also

Functions:

- [“UPCASE Function” on page 314](#)

MAD Function

Returns the median absolute deviation from the median.

Category: Descriptive Statistics

Returned data type: DOUBLE

Syntax

MAD(*expression-1* [, ...*expression-n*])

Arguments

expression

specifies any valid expression that evaluates to a numeric value of which the median absolute deviation from the median is to be computed.

Data type DOUBLE

Details

If all arguments have missing or null values, the result is a missing or null value. Otherwise, the result is the median absolute deviation from the median of the nonmissing or non-null values. The formula for the median is the same as the one that is used in the UNIVARIATE procedure. For more information, see Base SAS Procedures Guide: Statistical Procedures.

Example

The following statement illustrates the MAD function:

Statements	Results
select mad(2, 4, 1, 3, 5, 999999);	1.5

See Also

Functions:

- [“IQR Function” on page 247](#)
- [“MEDIAN Function” on page 270](#)
- [“PCTL Function” on page 279](#)

MAKEDATE Function

Returns the date as year, month, and day.

Category: Date and Time

Returned data type: DATE

Syntax

MAKEDATE(*y*, *m*, *d*)

Arguments

y
specifies the year.

m
specifies the month.

d
specifies the day.

Example

The following statements illustrates the MAKEDATE function:

Statements	Results
<pre>select makedate(2013, 10, 30);</pre>	30OCT2013
<pre>select makedate(y, m, d) from adate;</pre>	30OCT2013

See Also

Functions:

- [“DATEPART Function” on page 230](#)
- [“MAKETIME Function” on page 260](#)
- [“MAKETIMESTAMP Function” on page 261](#)
- [“TIMEPART Function” on page 311](#)

MAKETIME Function

Returns the time as hours, minutes, and seconds.

Category: Date and Time

Returned data type: TIME

Syntax

MAKETIME(*h, m, s*)

Arguments

- h***
specifies the hour.
- m***
specifies the minute.
- s***
specifies the second.

Example

The following statement illustrates the MAKETIME function:

Statements	Results
<code>select maketime(15,39,10);</code>	15:39:10
<code>select maketime(h, mn, s) from adate;</code>	15:39:10

See Also

Functions:

- [“DATEPART Function” on page 230](#)
- [“MAKEDATE Function” on page 260](#)
- [“MAKETIMESTAMP Function” on page 261](#)
- [“TIMEPART Function” on page 311](#)

MAKETIMESTAMP Function

Returns the timestamp.

Category: Date and Time

Returned data type: timestamp

Syntax

MAKETIMESTAMP(*d,t*)

MAKETIMESTAMP(*y,m,d,h,m,s*)

Arguments***d,t****d*
specifies the date.*t*
specifies the time.***y,m,d,h,m,s****y*
specifies the year.*m*
specifies the month.*d*
specifies the day.*h*
specifies the hours.*m*
specifies the minutes.*s*
specifies the seconds.**Example**

The following statements illustrate the MAKETIMESTAMP function:

Statements	Results
<code>select maketimestamp(date '2013-10-30', 30OCT2013:10:39:10 time '10:39:10');</code>	
<code>select maketimestamp(2013,10,30,13,39,10) 30OCT2013:13:39:10</code>	
<code>select maketimestamp(makedate(2013, 10, 3) 30OCT2013:13:39:10</code>	
<code>select maketimestamp(y, m, d, h, mn, s) f 30OCT2013:13:39:10</code>	

See Also**Functions:**

- [“DATEPART Function” on page 230](#)
- [“MAKEDATE Function” on page 260](#)
- [“MAKETIME Function” on page 260](#)
- [“TIMEPART Function” on page 311](#)

MARGRCLPRC Function

Calculates call prices for European options on stocks, based on the Margrabe model.

Category: Financial

Returned data type: DOUBLE

Syntax

MARGRCLPRC(X_1 , t , X_2 , *sigma1*, *sigma2*, *rho12*)

Arguments

X_1

is a nonmissing, positive value that specifies the price of the first asset.

Requirement Specify X_1 and X_2 in the same units.

Data type DOUBLE

t

is a nonmissing value that specifies the time to expiration, in years.

Data type DOUBLE

X_2

is a nonmissing, positive value that specifies the price of the second asset.

Requirement Specify X_2 and X_1 in the same units.

Data type DOUBLE

sigma1

is a nonmissing, positive fraction that specifies the volatility of the first asset.

Data type DOUBLE

sigma2

is a nonmissing, positive fraction that specifies the volatility of the second asset.

Data type DOUBLE

rho12

specifies the correlation between the first and second assets, $\rho_{x_1 x_2}$.

Range between -1 and 1

Data type DOUBLE

Details

The MARGRCLPRC function calculates the call price for European options on stocks, based on the Margrabe model. The function is based on the following relationship:

$$CALL = X_1 N(d_1) - X_2 N(d_2)$$

Arguments

X_1

specifies the price of the first asset.

X_2

specifies the price of the second asset.

N

specifies the cumulative normal density function.

$$d_1 = \frac{\left(\ln \left(\frac{X_1}{X_2} \right) + \left(\frac{\sigma^2}{2} \right) t \right)}{\sigma \sqrt{t}}$$

$$d_2 = d_1 - \sigma \sqrt{t}$$

$$\sigma^2 = \sigma_{x_1}^2 + \sigma_{x_2}^2 - 2\rho_{x_1, x_2} \sigma_{x_1} \sigma_{x_2}$$

The following arguments apply to the preceding equation:

t

specifies the time to expiration.

$\sigma_{x_1}^2$

specifies the variance of the first asset.

$\sigma_{x_2}^2$

specifies the variance of the second asset.

σ_{x_1}

specifies the volatility of the first asset.

σ_{x_2}

specifies the volatility of the second asset.

ρ_{x_1, x_2}

specifies the correlation between the first and second assets.

For the special case of $t=0$, the following equation is true:

$$CALL = \max((X_1 - X_2), 0)$$

Note: This function assumes that there are no dividends from the two assets.

For information about the basics of pricing, see “Using Pricing Functions” in Chapter 1 of *SAS Functions and CALL Routines: Reference*.

Comparisons

The MARGRCLPRC function calculates the call price for European options on stocks, based on the Margrabe model. The MARGRPTPRC function calculates the put price for

European options on stocks, based on the Margrabe model. These functions return a scalar value.

Example

The following statements illustrate the MARGRCLPRC function:

Statements	Results
<pre>select margrclprc(15, .5, 13, .06, .05, 1); put a;</pre>	2
<pre>select margrclprc(2, .25, 1, .3, .2, 1); put b;</pre>	1

See Also

Functions:

- [“MARGRPTPRC Function” on page 265](#)

MARGRPTPRC Function

Calculates put prices for European options on stocks, based on the Margrabe model.

Category:	Financial
Returned data type:	DOUBLE

Syntax

MARGRPTPRC(*X₁*, *t*, *X₂*, *sigma1*, *sigma2*, *rho12*)

Arguments

<i>X₁</i>	is a nonmissing, positive value that specifies the price of the first asset.
Requirement	Specify <i>X₁</i> and <i>X₂</i> in the same units.
Data type	DOUBLE
<i>t</i>	is a nonmissing value that specifies the time to expiration, in years.
Data type	DOUBLE
<i>X₂</i>	is a nonmissing, positive value that specifies the price of the second asset.
Requirement	Specify <i>X₂</i> and <i>X₁</i> in the same units.

Data type DOUBLE

sigma1

is a nonmissing, positive fraction that specifies the volatility of the first asset.

Data type DOUBLE

sigma2

is a nonmissing, positive fraction that specifies the volatility of the second asset.

Data type DOUBLE

rho12

specifies the correlation between the first and second assets, $\rho_{x_1 y_1}$

Range between -1 and 1

Data type DOUBLE

Details

The MARGRPTPRC function calculates the put price for European options on stocks, based on the Margrabe model. The function is based on the following relationship:

$$PUT = X_2 N(pd_1) - X_1 N(pd_2)$$

Arguments

X_1

specifies the price of the first asset.

X_2

specifies the price of the second asset.

N

specifies the cumulative normal density function.

$$pd_1 = \frac{\left(\ln \left(\frac{X_1}{X_2} \right) + \left(\frac{\sigma^2}{2} \right) t \right)}{\sigma \sqrt{t}}$$

$$pd_2 = pd_1 - \sigma \sqrt{t}$$

$$\sigma^2 = \sigma_{x_1}^2 + \sigma_{x_2}^2 - 2\rho_{x_1, x_2} \sigma_{x_1} \sigma_{x_2}$$

The following arguments apply to the preceding equation:

t

is a nonmissing value that specifies the time to expiration, in years.

$\sigma_{x_1}^2$

specifies the variance of the first asset.

$\sigma_{x_2}^2$

specifies the variance of the second asset.

σ_{x_1}

specifies the volatility of the first asset.

 σ_{x_2}

specifies the volatility of the second asset.

 ρ_{x_1, x_2}

specifies the correlation between the first and second assets.

To view the corresponding CALL relationship, see the “[MARGRCLPRC Function](#)” on [page 263](#).

For the special case of $t=0$, the following equation is true:

$$PUT = \max((X_2 - X_1), 0)$$

Note: This function assumes that there are no dividends from the two assets.

For information about the basics of pricing, see “Using Pricing Functions” in Chapter 1 of *SAS Functions and CALL Routines: Reference*.

Comparisons

The MARGRPTPRC function calculates the put price for European options on stocks, based on the Margrabe model. The MARGRCLPRC function calculates the call price for European options on stocks, based on the Margrabe model. These functions return a scalar value.

Example

The following statements illustrate the MARGRPTPRC function:

Statements	Results
<pre>select margrptprc(2, .25, 3, .06, .2, 1);</pre>	1.0000000000973
<pre>select margrptprc(3, .25, 4, .05, .3, 1);</pre>	1.00157624907712

See Also

Functions:

- “[MARGRCLPRC Function](#)” on [page 263](#)

MAX Function

Returns the maximum value in a column.

Categories: Aggregate
Descriptive Statistics

Returned data type: The same data type as the expression

Syntax

MAX(*expression*)

Arguments

expression
specifies any valid SQL expression.

Data type All data types are supported.

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The MAX function ignores null values and SAS missing values.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Comparisons

The MIN function returns the minimum value in a column. The MAX function returns the maximum value in a column.

Example

Table: [DENSITIES on page 539](#)

The following statement illustrates the MAX function.

Statements	Results
<pre>select max(population) from densities;</pre>	34248705

See Also

Functions:

- [“MIN Function” on page 271](#)

SELECT Statement Clauses:

- [“SELECT Clause” on page 389](#)
- [“GROUP BY Clause” on page 400](#)
- [“HAVING Clause” on page 401](#)

MEAN Function

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

Category: Descriptive Statistics

Returned data type: DOUBLE

Syntax

MEAN(*expression-1* [, ... *expression-n*])

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Requirement At least one non-null or nonmissing argument is required. Otherwise, the function returns a null or missing value.

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Comparisons

The GEOMEAN function returns the geometric mean, the HARMEAN function returns the harmonic mean, whereas the MEAN function returns the arithmetic mean (average).

Example

The following statements illustrate the MEAN function:

Statements	Results
<code>select mean(2,...,6);</code>	4
<code>select mean(1,2,3,2);</code>	2

See Also

Functions:

- [“GEOMEAN Function” on page 239](#)
- [“GEOMEANZ Function” on page 241](#)
- [“HARMEAN Function” on page 242](#)
- [“HARMEANZ Function” on page 243](#)

- [“MEDIAN Function” on page 270](#)

MEDIAN Function

Returns the median value.

Category: Descriptive Statistics

Returned data type: DOUBLE

Syntax

MEDIAN(*expression-1* [, ... *expression-n*])

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The MEDIAN function returns the median of the nonmissing or nonnull values. If all arguments have missing or null values, the result is a missing or null value.

Note: The formula that is used in the MEDIAN function is the same as the formula that is used in PROC UNIVARIATE in Base SAS Procedures Guide: Statistical Procedures. For more information, see SAS Elementary Statistics Procedures.

Comparisons

The MEDIAN function returns the median of nonmissing or nonnull values, whereas the MEAN function returns the arithmetic mean (average).

Example

The following statements illustrate the MEDIAN function:

Statements	Results
<pre>select median(2, 4, 1, 3);</pre>	2.5
<pre>select median(5, 8, 0, 3, 4);</pre>	4

See Also

Functions:

- [“MEAN Function” on page 269](#)

MIN Function

Returns the minimum value in an expression.

Categories: Aggregate
Descriptive Statistics

Returned data type: The same data type as the expression

Syntax

MIN(*expression*)

Arguments

expression
specifies any valid SQL expression.

Data type All data types are supported.

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The MIN function ignores null values and SAS missing values.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Comparisons

The MAX function returns the maximum value in a column. The MIN function returns the minimum value in a column.

Example

Table: [DENSITIES on page 539](#)

The following statement illustrates the MIN function.

Statements	Results
<code>select min(density) from densities;</code>	6.154657

See Also

Functions:

- [“MAX Function” on page 267](#)

SELECT Statement Clauses:

- [“SELECT Clause” on page 389](#)
- [“GROUP BY Clause” on page 400](#)
- [“HAVING Clause” on page 401](#)

MINUTE Function

Returns the minute from a time or datetime value.

Category: Date and Time

Returned data type: TINYINT

Syntax

MINUTE(*time* | *datetime*)

Arguments

time

specifies any valid expression that represents a time value.

Data type TIME

See [“Overview of Expressions and Predicates” on page 321](#)

datetime

specifies any valid expression that represents a datetime value.

Data type TIMESTAMP

See [“Overview of Expressions and Predicates” on page 321](#)

Details

In Hive, when the MINUTE function is used to search for a specific value, the function will fail if there is more than one column of type TIME in the table.

Example

Table: [CUSTONLINE on page 538](#)

The following statement illustrates the MINUTE function:

Statements	Results
<code>select minute(endtime) from custonline;</code>	5 21 4 25 47 6 5 15 35 6
<code>select minute(current_time);</code>	16
<code>select minute(localtimestamp);</code>	16

See Also

- “Dates and Times in FedSQL” on page 51

Functions:

- “DAY Function” on page 231
- “HOUR Function” on page 245
- “MONTH Function” on page 275
- “SECOND Function” on page 295
- “YEAR Function” on page 318

MOD Function

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

Category: Mathematical

Returned data type: DOUBLE

Syntax

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

Arguments

expression-1

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

Data type DOUBLE

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

expression-2

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

Restriction *expression-2* cannot be 0

Data type DOUBLE

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

Details

The MOD function returns the remainder from the division of *expression-1* by *expression-2*. When the result is non-zero, the result has the same sign as the first argument. The sign of the second argument is ignored.

The computation that is performed by the MOD function is exact if both of the following conditions are true:

- Both arguments are exact integers.
- All integers that are less than either argument have exact 8-byte floating-point representations.

If either of the above conditions is not true, a small amount of numerical error can occur in the floating-point computation. In this case

- MOD returns zero if the remainder is very close to zero or very close to the value of the second argument.
- MOD returns a missing value if the remainder cannot be computed to a precision of approximately three digits or more. In this case, SAS also writes an error message to the log.

Example

The following statements illustrate the MOD function:

Statements	Results
<code>select mod(10,3);</code>	1
<code>select mod(.3,-.1);</code>	0.1

Statements	Results
<code>select mod(1.7, .1);</code>	0.1
<code>select mod(.9, .3);</code>	5.55E-17

MONTH Function

Returns the numeric month from a date or datetime value.

Category: Date and Time

Returned data type: TINYINT

Syntax

MONTH(*date* | *datetime*)

Arguments

date

specifies any valid expression that represents a date value.

Data type DATE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

datetime

specifies any valid expression that represents a datetime value.

Data type TIMESTAMP

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Example

Table: [CUSTONLINE on page 538](#)

The following statement illustrates the MONTH function:

Statements	Results
select month(endtime) from custonline;	9
	10
	10
	11
	12
	1
	1
	2
	3
	3
select month(current_time);	10

See Also

- [“Dates and Times in FedSQL” on page 51](#)

Functions:

- [“DAY Function” on page 231](#)
- [“HOUR Function” on page 245](#)
- [“MINUTE Function” on page 272](#)
- [“SECOND Function” on page 295](#)
- [“YEAR Function” on page 318](#)

NMISS Function

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

Category: Aggregate

Returned data type: BIGINT

Syntax

NMISS(*expression*)

Arguments

expression

specifies any valid SQL expression.

Data type CHAR, DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

NMISS indicates the total number of null or SAS missing values.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Example

Table: [WORLDTEMPS on page 542](#)

The following statement illustrates the NMISS function:

Statements	Results
<pre>select nmiss(AvgHigh) from worldtemps;</pre>	1

See Also

- “How FedSQL Processes Nulls and SAS Missing Values” on page 18

SELECT Statement Clauses:

- “SELECT Clause” on page 389
- “GROUP BY Clause” on page 400
- “HAVING Clause” on page 401

OCTET_LENGTH Function

Returns the number of bytes in a string of any data type.

Category: Character

Returned data type: BIGINT

Syntax

OCTET_LENGTH(*expression*)

Arguments

expression

specifies any valid expression.

Data type All data types are valid.

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

Details

The OCTET_LENGTH function counts a multibyte character as multiple characters. For example, if the specified string contains 6 two-byte characters, the OCTET_LENGTH function returns a value of 12.

If the character string is a bit string, the OCTET_LENGTH function returns the number of bytes it takes to hold the number of bits.

Comparisons

The CHARACTER_LENGTH function also returns the number of characters in a character string, but counts a multibyte character as a single character. The OCTET_LENGTH function counts a multibyte character as multiple characters.

Example

The following statements illustrate the OCTET_LENGTH function:

Statements	Results
select octet_length('December');	8
select octet_length(B'0100011100');	2
select octet_length(x'FEA364BEA234');	6
select octet_length(' FedSQL ');	18

See Also

Functions:

- [“CHARACTER_LENGTH Function” on page 215](#)

ORDINAL Function

Orders a list of values, and returns a value that is based on a position in the list.

Category:	Descriptive Statistics
Returned data type:	DOUBLE

Syntax

ORDINAL(*position*, *expression-1*, *expression-2* [, ...*expression-n*])

Arguments

position

specifies an integer that is less than or equal to the number of elements in the list of arguments.

Requirement *position* must be a positive number.

Data type DOUBLE

expression

specifies any valid expression that evaluates to a numeric value.

Requirement At least two arguments are required.

Data type DOUBLE

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

Details

The ORDINAL function sorts the list and returns the argument in the list that is specified by *position*. Missing values are sorted low and are placed before any numeric values.

Comparisons

The ORDINAL function counts both null, missing, non-null, and nonmissing values, whereas the SMALLEST function counts only non-null and nonmissing values.

Example

The following statement illustrates the ORDINAL function:

Statement	Result
<code>select ordinal(4,1,.,2,3,-4,5,6,7);</code>	2

PCTL Function

Returns the percentile that corresponds to the percentage.

Category: Descriptive Statistics

Returned data type: DOUBLE

Syntax

PCTL[*n*](*percentage*, *expression*[, ...*expression*])

Arguments

n

is a digit from 1 to 5 that specifies the definition of the percentile to be computed.

Default definition 5

Data type DOUBLE

See QNTLDEF= (alias PCTLDEF=) descriptions in the table “Methods for Computing Quantile Statistics” in the *Base SAS Procedures Guide*.

percentage

specifies the percentile to be computed.

Data type DOUBLE

Tip *percentage* is numeric where, $0 \leq \textit{percentage} \leq 100$.

expression

specifies any valid expression that evaluates to a numeric value, whose value is computed in the percentile calculation.

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The PCTL function returns the percentile of the non-null or nonmissing values corresponding to the percentage. If *percentage* is null or missing, less than zero, or greater than 100, the PCTL function generates an error message.

Example

The following statements illustrate the PCTL function:

Description	Statement	Result
Lower quartile	<code>select pctl(25,2,4,1,3);</code>	1.5
Percentile definition two	<code>select pctl2(25,2,4,1,3);</code>	1
Lower tertile	<code>select pctl(100/3,2,4,1,3);</code>	2
Percentile definition three	<code>select pctl3(100/3,2,4,1,3);</code>	2
Median	<code>select pctl(50,2,4,1,3);</code>	2.5
Upper tertile	<code>select pctl(200/3,2,4,1,3);</code>	3
Upper quartile	<code>select pctl(75,2,4,1,3);</code>	3.5

PI Function

Returns the constant value of PI as a floating-point value.

Category: Mathematical

Returned data type: DOUBLE

Syntax

PI()

Details

The value of PI is 3.141593.

Example

The following statement illustrates the PI function:

Statements	Results
<code>select pi();</code>	3.141593

POWER Function

Returns the value of a numeric value expression raised to a specified power.

Category: Mathematical

Returned data type: DOUBLE

Syntax

POWER(*numeric-expression*, *power-numeric-expression*)

Required Arguments

numeric-expression

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

Data type BIGINT, DOUBLE, FLOAT, INTEGER, REAL, SMALLINT, TINYINT

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

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

Details

If *numeric-expression* is a null or missing value, then the POWER function returns a null or missing value. If the result is a number that does not fit into the range of a DOUBLE, the POWER function fails.

Example

Table: [DENSITIES on page 539](#)

The following statements illustrate the POWER function:

Statements	Results
<code>select power(5*3, 2);</code>	225
<code>select name from densities where squaremiles >= power(density,2);</code>	Afghanistan Algeria Angola Argentina Australia

PROBBNML Function

Returns the probability from a binomial distribution.

Category:	Probability
Returned data type:	DOUBLE

Syntax

PROBBNML(*p*, *n*, *m*)

Arguments

<i>p</i>	is a numeric probability of success parameter.
Range	$0 \leq p \leq 1$
Data type	DOUBLE
<i>n</i>	is an integer number of independent Bernoulli trials parameter.
Range	$n > 0$

Data type INTEGER

m

is an integer number of successes random variable.

Range $0 \leq m \leq n$

Data type INTEGER

Details

The PROBBNML function returns the probability that an observation from a binomial distribution, with probability of success p , number of trials n , and number of successes m , is less than or equal to m . To compute the probability that an observation is equal to a given value m , compute the difference of two probabilities from the binomial distribution for m and $m-1$ successes.

Example

The following statement illustrates the PROBBNML function:

Statements	Results
<pre>select probbnml(0.5,10,4);</pre>	0.376953125

PROBBNRM Function

Returns a probability from a bivariate normal distribution.

Category: Probability

Returned data type: DOUBLE

Syntax

PROBBNRM(*x*, *y*, *r*)

Arguments

x

specifies a numeric constant, variable, or expression.

Data type DOUBLE

y

specifies a numeric constant, variable, or expression.

Data type DOUBLE

r

is a numeric correlation coefficient.

Range	$-1 \leq r \leq 1$
Data type	DOUBLE

Details

The PROBBNRM function returns the probability that an observation (X, Y) from a standardized bivariate normal distribution with mean 0, variance 1, and a correlation coefficient r , is less than or equal to (x, y) . That is, it returns the probability that $X \leq x$ and $Y \leq y$. The following equation describes the PROBBNRM function, where u and v represent the random variables x and y , respectively:

$$PROBBNRM(x, y, r) = \frac{1}{2 \pi \sqrt{1 - r^2}} \int_{-\infty}^x \int_{-\infty}^y \exp \left[-\frac{u^2 - 2ruv + v^2}{2(1 - r^2)} \right] dv du$$

Example

The following statement illustrates the PROBBNRM function:

Statements	Results
<pre>select probbnrm(.4, -.3, .2);</pre>	0.27831833451902

PROBCHI Function

Returns the probability from a chi-square distribution.

Category:	Probability
Returned data type:	DOUBLE

Syntax

PROBCHI(*x*, *df*[, *nc*])

Arguments

x
is a numeric random variable.

Range	$x \geq 0$
Data type	DOUBLE

df
is a numeric degrees of freedom parameter.

Range	$df > 0$
Data type	DOUBLE

nc

is an optional numeric noncentrality parameter.

Range $nc \geq 0$ **Data type** DOUBLE

Details

The PROBCHI function returns the probability that an observation from a chi-square distribution, with degrees of freedom *df* and noncentrality parameter *nc*, is less than or equal to *x*. This function accepts a noninteger degrees of freedom parameter *df*. If the optional parameter *nc* is not specified or has the value 0, the value returned is from the central chi-square distribution.

Example

The following statement illustrates the PROBCHI function:

Statements	Results
<pre>select probchi(11.264,11);</pre>	0.5785813293173

PROBT Function

Returns the probability from a t distribution of the values in an expression.

Categories: Aggregate
Descriptive Statistics

Returned data type: DOUBLE

Syntax

PROBT(*expression*)

Arguments

expression
specifies any valid SQL expression.

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The PROBT function returns the probability that an observation from a Student's t distribution, with degrees of freedom $n-1$ and noncentrality parameter nc equal to 0, is less than or equal to *expression*.

The significance level of a two-tailed t test is given by this code line.

```
p=(1-probt(abs(x),df))*2;
```

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Comparisons

The STUDENTS_T function returns the Student's t-distribution. The PROBT function returns the probability that the Student's t-distribution is less than or equal to a given value.

Example

Table: [DENSITIES on page 539](#)

The following statements illustrate the PROBT function:

Statements	Results
<code>select probt(density) from densities;</code>	0.006068
<code>select probt(population) from densities;</code>	0.009722

See Also

Functions:

- “STUDENTS_T Function” on page 305

SELECT Statement Clauses:

- “SELECT Clause” on page 389
- “GROUP BY Clause” on page 400
- “HAVING Clause” on page 401

PUT Function

Returns a value using a specified format.

Category: Special

Returned data type: NVARCHAR VARBINARY BINARY

Syntax

PUT(*source*,*format*.)

Arguments

source

identifies the variable or constant whose value you want to reformat.

Data type a data type that is supported by the format argument

format.

contains the SAS or FedSQL format that you want applied to the variable or constant that is specified in the source.

To override the default alignment, you can add an alignment specification to a format:

- L
left aligns the value
- C
centers the value
- R
right aligns the value

Restriction the format must be the same type as the value of *source*

Details

The formats that are supported for SAS and third-party data sources differ. For more information, see [“How to Format Output with the PUT Function” on page 71](#).

The result of the PUT function is always a character string. If the source is numeric, the resulting string is right aligned. If the source is character, the result is left aligned.

Use PUT to format constants and to output stored data in a different format. Use PUT to convert a numeric value to a character value.

Comparisons

The CAST function permanently modifies the data type of the specified input variable. The PUT function affects the output of the query in which it is specified.

Example

Table: [WORLDTEMPS on page 542](#)

The following statement illustrates the PUT function.

Statement	Result
<pre>select put(0.6666666667, fract8.);</pre>	2/3

Statement	Result
<code>select put (date(), date.);</code>	19SEP13
<code>select put(AvgLow, 4.1) from worldtemps;</code>	45.0 33.0 17.0 68.0 56.0 57.0 28.0 51.0 75.0 36.0 33.0 25.0

See Also

[“How to Format Output with the PUT Function” on page 71](#)

QTR Function

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

Category: Date and Time

Returned data type: DOUBLE

Syntax

QTR(*date*)

Arguments

date

specifies any valid expression that represents a SAS date value.

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The QTR function returns a value of 1, 2, 3, or 4 from a SAS date value to indicate the quarter of the year in which a date value falls.

For more information about how FedSQL handles date and time values, see [“Dates and Times in FedSQL” on page 51](#).

Example

The following statements illustrate the QTR function:.

Statements	Results
<code>select qtr(16983);</code>	3
<code>select qtr(17075);</code>	4

See Also

Functions:

- [“YYQ Function” on page 319](#)

QUOTE Function

Adds double quotation marks to a character value.

Category: Character

Returned data type: NCHAR

Syntax

QUOTE(*expression*)

Arguments

expression

specifies any valid expression that evaluates to a character string.

Data type NCHAR

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The QUOTE function adds double quotation marks, the default character, to a character value. If double quotation marks are found within the argument, they are doubled in the output.

The length of the receiving variable must be long enough to contain the argument (including trailing blanks), leading and trailing quotation marks, and any embedded quotation marks that are doubled. For example, if the argument is ABC followed by three trailing blanks, then the receiving variable must have a length of at least eight to hold “ABC###”. (The character # represents a blank space.) If the receiving field is not long enough, the QUOTE function returns a blank string, and writes an invalid argument note to the SAS log.

A string of characters enclosed in double quotation marks is a DS2 identifier and not a character constant. The double quotation marks become part of the identifier. Quoted identifiers cannot be used to create column names in an output table.

Example

The following statements illustrate the QUOTE function:

Statements	Results
<code>select quote('A"B');</code>	<code>"A""B"</code>
<code>select quote('A''B');</code>	<code>"A'B"</code>
<code>select quote('Paul''s Catering Service');</code>	<code>"Paul's Catering Service"</code>

RADIANS Function

Returns the number of radians converted from a numeric degree value.

Category: Trigonometric

Returned data type: DOUBLE

Syntax

RADIANS(*expression*)

Arguments

expression

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

Data type BIGINT, DOUBLE, FLOAT, INTEGER, REAL, SMALLINT, TINYINT

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Example

The following statement illustrate the RADIANS function:

Statements	Results
<code>select radians(360);</code>	<code>6.283185</code>

See Also

Functions:

- [“DEGREES Function” on page 232](#)

RANGE Function

Returns the range between values in an expression.

Categories: Aggregate
Descriptive Statistics

Returned data type: the same type as the expression

Syntax

RANGE(*expression*)

Arguments

expression
specifies any valid SQL expression.

Data type BIGINT, DOUBLE, FLOAT, INTEGER, REAL, SMALLINT, TINYINT

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The RANGE function returns the difference between the largest and the smallest values in the specified expression. The RANGE function ignores null values and SAS missing values.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Example

Table: [WORLDTEMPS on page 542](#)

The following statement illustrates the RANGE function:

Statements	Results
<code>select range(AvgHigh) from worldtemps;</code>	27

See Also

SELECT Statement Clauses:

- [“SELECT Clause” on page 389](#)
- [“GROUP BY Clause” on page 400](#)
- [“HAVING Clause” on page 401](#)

REPEAT Function

Repeats a character expression.

Category: Character

Returned data type: VARCHAR, NVARCHAR

Syntax

REPEAT(*expression*, *n*)

Arguments

expression

specifies any valid expression that evaluates to a character string.

Data type CHAR, NCHAR

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

n

specifies the number of times to repeat *expression*.

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

Data type INTEGER

Details

The REPEAT function returns a character value consisting of the first argument repeated *n* times. Thus, the first argument appears *n*+1 times in the result.

Example

The following statement illustrates the REPEAT function:

Statements	Results
<code>select repeat('ONE',2);</code>	ONEONEONE

REVERSE Function

Reverses a character expression.

Category: Character

Returned data type: NCHAR

Syntax

REVERSE(*expression*)

Arguments

expression

specifies any valid expression that evaluates to a character string.

Data type NCHAR

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The REVERSE function returns a character value with the last character in the expression is the first character in the result, the next-to-last character in the expression is the second character in the result, and so on.

Note: Trailing blanks in the expression become leading blanks in the result.

Example

The following statement illustrates the REVERSE function:

Statements	Results
	-----1
<code>select reverse('xyz ');</code>	zyx

RMS Function

Returns the root mean square.

Category:	Descriptive Statistics
Returned data type:	DOUBLE

Syntax

RMS(*expression* [, ...*expression*])

Arguments

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

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The root mean square is the square root of the arithmetic mean of the squares of the values. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the root mean square of the non-null or nonmissing values.

Let *n* be the number of arguments with non-null or nonmissing values, and let *x*₁, *x*₂, ..., *x*_{*n*} be the values of those arguments. The root mean square is calculated as follows.

$$\sqrt{\frac{x_1^2 + x_2^2 + \dots + x_n^2}{n}}$$

Example

The following statements illustrate the RMS function:

Statements	Results
<code>select rms(1,7);</code>	5
<code>select rms(.,1,5,11);</code>	7

SECOND Function

Returns the second from a time or datetime value.

Category: Date and Time

Returned data type: DOUBLE

Syntax

SECOND(*time* | *datetime*)

Arguments

time

specifies any valid expression that represents a time value.

Data type TIME

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

datetime

specifies any valid expression that represents a datetime value.

Data type TIMESTAMP

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

In Hive, when the SECOND function is used to search for a specific value, the function will fail if there is more than one column of type TIME in the table.

Example

Table: [CUSTONLINE on page 538](#)

The following statement illustrates the SECOND function:

Statements	Results
<code>select second(endtime) from custonline;</code>	1.253 9.421 55.746 9.398 45.221 15.766 56.288 33.955 27.908 20.475

See Also

- [“Dates and Times in FedSQL” on page 51](#)

Functions:

- [“DAY Function” on page 231](#)
- [“HOUR Function” on page 245](#)
- [“MINUTE Function” on page 272](#)
- [“MONTH Function” on page 275](#)
- [“YEAR Function” on page 318](#)

SIGN Function

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

Category: Mathematical

Returned data type: TINYINT

Syntax

SIGN(*expression*)

Arguments

expression

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

Data type BIGINT, DOUBLE, FLOAT, INTEGER, REAL, SMALLINT, TINYINT

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The SIGN function returns the following values:

-1
if *expression* < 0

0
if *expression* = 0

1
if *expression* > 0

Example

The following statements illustrate the SIGN function:

Statements	Results
<code>select sign(-5);</code>	-1
<code>select sign(5);</code>	1
<code>select sign(0);</code>	0

SIN Function

Returns the trigonometric sine.

Category: Trigonometric

Returned data type: DOUBLE

Syntax

`SIN(expression)`

Arguments

expression

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

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Example

The following statements illustrate the SIN function:

Statements	Results
<code>select sin(25.6);</code>	0.450441
<code>select sin(5);</code>	-0.95892

See Also

Functions:

- [“ACOS Function” on page 191](#)
- [“ASIN Function” on page 192](#)
- [“COS Function” on page 218](#)
- [“SINH Function” on page 298](#)

SINH Function

Returns the hyperbolic sine.

Category: Trigonometric

Returned data type: DOUBLE

Syntax

`SINH(expression)`

Arguments

expression

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

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The SINH function returns the hyperbolic sine of the argument, which is given by the following equation.

$$e^{\text{argument}} - e^{-\text{argument}} / 2$$

Example

The following statements illustrate the SINH function:

Statements	Results
<code>select sinh(0);</code>	0
<code>select sinh(1);</code>	1.175201
<code>select sinh(-1.0);</code>	-1.1752

See Also

Functions:

- [“COSH Function” on page 219](#)
- [“SIN Function” on page 297](#)
- [“TANH Function” on page 310](#)

SKEWNESS Function

Returns the skewness of all values in an expression.

Categories: Aggregate
Descriptive Statistics

Returned data type: DOUBLE

Syntax

SKEWNESS(*expression*)

Arguments

expression

specifies any valid SQL expression.

Interaction At least three valid values are required in the column to perform the calculation. Otherwise the function returns a null value.

Data type DOUBLE

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

Details

Skewness is a measure of the tendency of the deviations from the mean to be larger in one direction than in the other. A positive value for skewness indicates that the data is skewed to the right. A negative value indicates that the data is skewed to the left.

Null values and SAS missing values are ignored and are not included in the computation.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Example

Table: [WORLDTEMPS](#) on page 542

The following statement illustrates the SKEWNESS function:

Statements	Results
<code>select skewness(AvgHigh) from worldtemps;</code>	-0.69811

See Also

Functions:

- “STDDEV Function” on page 303

SELECT Statement Clauses:

- “SELECT Clause” on page 389
- “GROUP BY Clause” on page 400
- “HAVING Clause” on page 401

SMALLEST Function

Returns the *k*th smallest non-null or nonmissing value.

Category: Descriptive Statistics

Returned data type: DOUBLE

Syntax

SMALLEST(*k*, *expression* [, ...*expression*])

Arguments

k

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

Data type DOUBLE

See “<sql-expression>” on page 339

[“FedSQL Expressions” on page 42](#)

expression

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

Data type DOUBLE

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

Details

If k is null or missing, less than zero, or greater than the number of values, the result is a null or missing value.

Comparisons

The SMALLEST function differs from the ORDINAL function in that SMALLEST ignores null and missing values, but ORDINAL counts null and missing values.

Example

The following statements illustrate the SMALLEST function:

Statements	Results
<code>select smallest(1, 456, 789, .Q, 123);</code>	123
<code>select smallest(2, 456, 789, .Q, 123);</code>	456
<code>select smallest(3, 456, 789, .Q, 123);</code>	789
<code>select smallest(4, 456, 789, .Q, 123);</code>	.

See Also

Functions:

- [“LARGEST Function” on page 251](#)
- [“ORDINAL Function” on page 278](#)
- [“PCTL Function” on page 279](#)

SQRT Function

Returns the square root of a value.

Category: Mathematical

Returned data type: DOUBLE

Syntax

SQRT(*expression*)

Arguments

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

Data type	DOUBLE
See	“<sql-expression>” on page 339 “FedSQL Expressions” on page 42

Example

The following statements illustrate the SQRT function:

Statements	Results
<code>select sqrt(36);</code>	6
<code>select sqrt(25);</code>	5
<code>select sqrt(4.4);</code>	2.097618

STD Function

Returns the standard deviation.

Categories:	Aggregate Descriptive Statistics
Returned data type:	DOUBLE

Syntax

STD(*expression-1*, *expression-2* [, ...*expression-n*])

Arguments

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

Requirement	At least two non-null or nonmissing arguments are required. Otherwise, the function returns a null or missing value.
--------------------	--

Data type	DOUBLE
See	“<sql-expression>” on page 339
	“FedSQL Expressions” on page 42.

Example

The following statements illustrate the STD function:

Statements	Results
<code>select std(2,6);</code>	2.82842712474619
<code>select std(2,6,.);</code>	2.82842714274619
<code>select std(2,4,6,3,1);</code>	1.92353840616714

STDDEV Function

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

Categories: Aggregate
Descriptive Statistics

Alias: STD

Returned data type: DOUBLE

Syntax

STDDEV(*expression*)

Arguments

expression

specifies any valid SQL expression.

Interaction At least two valid values are required in the column to perform the calculation. Otherwise the function returns a null value.

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The standard deviation is calculated as the square root of the variance.

Null values and SAS missing values are ignored and are not included in the computation.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns.

When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group.

FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Example

Table: [WORLDTEMPS on page 542](#)

The following statement illustrates the STDDEV function:

Statements	Results
<pre>select stddev(AvgHigh) from worldtemps;</pre>	7.811414

See Also

Functions:

- “STDERR Function” on page 304
- “VARIANCE Function” on page 316

SELECT Statement Clauses:

- “SELECT Clause” on page 389
- “GROUP BY Clause” on page 400
- “HAVING Clause” on page 401

STDERR Function

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

Categories: Aggregate
Descriptive Statistics

Returned data type: DOUBLE

Syntax

STDERR(*expression*)

Arguments

expression

specifies any valid SQL expression.

Interaction At least two valid values are required in the column to perform the calculation. Otherwise the function returns a null value.

Data type DOUBLE

Details

Null values and SAS missing values are ignored and are not included in the computation.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns.

When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group.

FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Example

Table: [WORLDTEMPS on page 542](#)

The following statement illustrates the STDERR function:

Statements	Results
<pre>select stderr(AvgHigh) from worldtemps;</pre>	2.35523

See Also

Functions:

- “STDDEV Function” on page 303

SELECT Statement Clauses:

- “SELECT Clause” on page 389
- “GROUP BY Clause” on page 400
- “HAVING Clause” on page 401

STUDENTS_T Function

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

Categories: Aggregate
Descriptive Statistics

Alias: T

Returned data type: DOUBLE

Syntax

STUDENTS_T(*expression*)

Arguments

expression
specifies any valid SQL expression.

Data type	DOUBLE
See	“<sql-expression>” on page 339
	“FedSQL Expressions” on page 42

Details

The STUDENTS_T function returns the probability for the Student's t distribution with $n-1$ degrees of freedom and a central distribution ($nc=0$).

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Comparisons

The STUDENTS_T function returns the Student's t distribution. The PROBT function returns the probability that the Student's t distribution is less than or equal to a given value.

Example

Table: [DENSITIES on page 539](#)

The following statements illustrate the STUDENTS_T function:

Statements	Results
<code>select students_t(density) from densities;</code>	3.565402
<code>select students_t(population) from densities;</code>	3.267438

See Also

Functions:

- [“PROBT Function” on page 285](#)

SELECT Statement Clauses:

- [“SELECT Clause” on page 389](#)

- “GROUP BY Clause” on page 400
- “HAVING Clause” on page 401

SUBSTRING Function

Extracts a substring from a character string.

Category: Character

Returned data type: VARCHAR NVARCHAR

Syntax

SUBSTRING(*character-string* FROM *position* [FOR *length*])

Arguments

character-string

specifies any valid expression that evaluates to a character string. *character-string* is the source string that is searched for a substring.

Data type CHAR, VARCHAR, NVARCHAR

position

specifies the beginning character position.

Interaction If *position* is larger than the length of the source string, FedSQL returns a null value. For example, if you ask for a substring starting at character five but the source string is only four characters long, you get a null result.

Data type INTEGER

length

specifies the length of the substring to extract.

Interaction If *length* is zero, a negative value, or larger than the length of the expression that remains in string after *position*, FedSQL extracts the remainder of the expression.

Data type INTEGER

Tip If you omit *length*, FedSQL extracts the remainder of the expression.

Details

The SUBSTRING function returns a portion of an expression that you specify in *character-string*. The portion begins with the character that you specify by *position*, and is the number of characters that you specify in *length*.

Example

Table: [AFEWORDS](#) on page 537

The following statement illustrates the SUBSTRING function.

Statements	Results
<pre>select substring(word2 from 2 for 3);</pre>	HER HIN ODY

See Also

Functions:

- [“TRIM Function” on page 313](#)

SUM Function

Returns the sum of all the values in an expression.

Categories:	Aggregate Descriptive Statistics
Returned data type:	The same data type as the expression

Syntax

SUM(*expression*)

Arguments

expression
specifies any valid SQL expression.

Requirement	The result of the SUM function must be within the range of the data type.
Data type	BIGINT, DOUBLE, FLOAT, INTEGER, REAL, SMALLINT, TINYINT
See	“<sql-expression>” on page 339 “FedSQL Expressions” on page 42

Details

Null values and SAS missing values are ignored and are not included in the computation. You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group.

FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Example

Table: [DENSITIES on page 539](#)

The following statements illustrate the SUM function:

Statements	Results
<code>select sum(density) from densities;</code>	1728.324
<code>select sum(population) from densities;</code>	1.2278E8

See Also

SELECT Statement Clauses:

- [“SELECT Clause” on page 389](#)
- [“GROUP BY Clause” on page 400](#)
- [“HAVING Clause” on page 401](#)

TAN Function

Returns the tangent.

Category:	Trigonometric
Returned data type:	DOUBLE

Syntax

TAN(*expression*)

Arguments

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

Restriction	<i>expression</i> cannot be an odd multiple of $\pi / 2$
Requirement	The expression must evaluate to a value in radians.
Data type	DOUBLE
See	“<sql-expression>” on page 339 “FedSQL Expressions” on page 42

Example

The following statements illustrate the TAN function:

Statements	Results
<code>select tan(0.5);</code>	0.546302
<code>select tan(3.14159/3);</code>	1.732047

See Also

Functions:

- [“ATAN Function” on page 193](#)
- [“ATAN2 Function” on page 194](#)
- [“COS Function” on page 218](#)
- [“COT Function” on page 220](#)
- [“SIN Function” on page 297](#)
- [“TANH Function” on page 310](#)

TANH Function

Returns the hyperbolic tangent.

Category: Trigonometric
Returned data type: DOUBLE

Syntax

TANH(*expression*)

Arguments

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

Restriction	<i>expression</i> cannot be an odd multiple of $\pi/2$
Data type	DOUBLE
See	“<sql-expression>” on page 339 “FedSQL Expressions” on page 42

Details

The TANH function returns the hyperbolic tangent of the argument, which is given by the following equation.

$$\frac{e^{\text{argument}} - e^{-\text{argument}}}{e^{\text{argument}} + e^{-\text{argument}}}$$

Example

The following statements are examples of the TANH function:

Statements	Results
<code>x=tanh(0);</code>	0
<code>x=tanh(0.5);</code>	0.46211715726
<code>select tanh(-0.5);</code>	-0.46212

See Also

Functions:

- [“ATAN Function” on page 193](#)
- [“ATAN2 Function” on page 194](#)
- [“COSH Function” on page 219](#)
- [“SINH Function” on page 298](#)
- [“TAN Function” on page 309](#)

TIMEPART Function

Returns the time as hours, minutes, and seconds.

Category: Date and Time

Returned data type: TIME

Syntax

TIMEPART(*ts*)

Arguments

ts
specifies the timestamp.

Example

The following statement illustrates the TIMEPART function:

Statements	Results
<pre>select timepart(timestamp '2013-10-30 15:39:10') 15:39:10</pre>	

See Also

Functions:

- [“DATEPART Function” on page 230](#)
- [“MAKEDATE Function” on page 260](#)
- [“MAKETIME Function” on page 260](#)
- [“MAKETIMESTAMP Function” on page 261](#)

TODAY Function

Returns the current date as a numeric SAS date value.

Category: Date and Time

Returned data type: DOUBLE

Syntax

TODAY()

Details

The TODAY function does not take any arguments. It produces the current date in the form of a SAS date value, which is the number of days since January 1, 1960.

For more information about how FedSQL handles dates, see [“Dates and Times in FedSQL” on page 51](#).

Example

The following statement illustrates the TODAY function:

Statements	Results
<pre>select today();</pre>	19612
<pre>select put(td,date.);</pre>	11SEP13

TRIM Function

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

Category: Character

Returned data type: VARCHAR NVARCHAR

Syntax

TRIM(**[BOTH | LEADING | TRAILING]** [*trim-character*] **FROM** *column*)

Arguments

BOTH | LEADING | TRAILING

specifies whether to remove the leading characters, the trailing characters, or both.

Default BOTH

trim-character

specifies one character to remove from *column*. Enclose a literal character in single quotation marks. If *trim-character* is not specified, the TRIM function trims all blank spaces, not just one character.

Default Blank

Data type CHAR, VARCHAR, NVARCHAR

column

is any valid expression that evaluates to a column name.

Details

The TRIM function is useful for trimming character strings of blanks or other characters before they are concatenated.

Example

Table: [AFEWWORDS](#) on page 537

The following statements illustrate the TRIM function:

Statements	Results
<code>select trim(word1) from afewwords;</code>	*some/ *every* *no*
<code>select trim(both '*' from word1) from afewwords;</code>	some every no

Statements	Results
<code>select trim(leading '*' from word1) from afewwords;</code>	some/ every* no*
<code>select trim(trailing '*' from word1) from afewwords;</code>	*some/ *every *no

See Also

Functions:

- [“SUBSTRING Function” on page 307](#)

UPCASE Function

Converts all letters in an argument to uppercase.

Category: Character

Alias: UPPER

Returned data type: VARCHAR, NVARCHAR

Syntax

UPCASE(*expression*)

Arguments

expression

specifies any valid expression that evaluates to a character string.

Data type CHAR, NCHAR

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The UPCASE function copies a character expression, converts all lowercase letters to uppercase letters, and returns the altered value as a result.

Comparisons

The LOWCASE function converts all letters in an argument to lowercase letters. The UPCASE function converts all letters in an argument to uppercase letters.

Example

The following statement illustrates the UPCASE function:

Statements	Results
<pre>select upcase('John B. Smith');</pre>	JOHN B. SMITH

See Also

Functions:

- [“LOWCASE Function” on page 258](#)

USS Function

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

Categories: Aggregate
Descriptive Statistics

Returned data type: DOUBLE

Syntax

USS(*expression*)

Arguments

expression
specifies any valid SQL expression.

Data type DOUBLE

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

Null values and SAS missing values are ignored and are not included in the computation.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns.

When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group.

FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Comparisons

The CSS function returns the corrected sum of squares of all values. The USS function returns the uncorrected sum of squares.

Example

Table: [DENSITIES on page 539](#)

The following statements illustrate the USS function:

Statements	Results
<code>select uss(density) from densities;</code>	510190
<code>select uss(population) from densities;</code>	2.82E15

See Also

Functions:

- [“CSS Function” on page 222](#)

SELECT Statement Clauses:

- [“SELECT Clause” on page 389](#)
- [“GROUP BY Clause” on page 400](#)
- [“HAVING Clause” on page 401](#)

VARIANCE Function

Returns the measure of the dispersion of all values in an expression.

Categories:	Aggregate Descriptive Statistics
Returned data type:	DOUBLE

Syntax

`VARIANCE(expression)`

Arguments

expression
specifies any valid SQL expression.

Interaction At least two values are required to perform the calculation. Otherwise the function returns a null value.

Data type	DOUBLE
See	“<sql-expression>” on page 339 “FedSQL Expressions” on page 42

Details

Null values and SAS missing values are ignored and are not included in the computation.

You can use an aggregate function to produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. The GROUP BY clause groups data by a specified column or columns. When you use a GROUP BY clause, the aggregate function in the SELECT clause or in a HAVING clause instructs FedSQL in how to summarize the data for each group. FedSQL calculates the aggregate function separately for each group. If GROUP BY is omitted, then all the rows in the table or view are considered to be a single group.

Example

Table: [DENSITIES on page 539](#)

The following statements illustrate the VARIANCE function:

Statements	Results
<code>select variance(density) from densities;</code>	23498.12
<code>select variance(population) from densities;</code>	1.412E14

See Also

Functions:

- [“STDDEV Function” on page 303](#)

SELECT Statement Clauses:

- [“SELECT Clause” on page 389](#)
- [“GROUP BY Clause” on page 400](#)
- [“HAVING Clause” on page 401](#)

WEEKDAY Function

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

Category: Date and Time

Returned data type: DOUBLE

Syntax

WEEKDAY(*expression*)

Arguments

expression
specifies any valid expression that represents a SAS date value.

Data type	DOUBLE
See	“<sql-expression>” on page 339
	“FedSQL Expressions” on page 42

Details

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

For information about how FedSQL handles date and times values, see [“Dates and Times in FedSQL” on page 51](#).

Example

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

Statements	Results
<pre>select weekday(today());</pre>	1

YEAR Function

Returns the year from a date or datetime value.

Category:	Date and Time
Returned data type:	SMALLINT

Syntax

YEAR(*date* | *datetime*)

Arguments

date
specifies any valid expression that represents a date value.

Data type	DATE
See	“<sql-expression>” on page 339

[“FedSQL Expressions” on page 42](#)

datetime

specifies any valid expression that represents a datetime value.

Data type **TIMESTAMP**

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Example

Table: [CUSTONLINE on page 538](#)

The following statement illustrates the YEAR function:

Statements	Results
select year(endtime) from custonline;	2012
	2012
	2012
	2012
	2012
	2013
	2013
	2013
	2013
	2013
	2013
select year(current_time);	2013

See Also

- [“Dates and Times in FedSQL” on page 51](#)

Functions:

- [“DAY Function” on page 231](#)
- [“HOUR Function” on page 245](#)
- [“MINUTE Function” on page 272](#)
- [“MONTH Function” on page 275](#)
- [“SECOND Function” on page 295](#)

YYQ Function

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

Category: Date and Time

Returned data type: DOUBLE

Syntax

YYQ(*year*, *quarter*)

Arguments

year
specifies any valid expression that evaluates to a two-digit or four-digit integer that represents the year.

Interaction The YEARCUTOFF= system option defines the year value for two-digit dates.

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” in *SAS DS2 Language Reference*

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

Data type DOUBLE

See Chapter 13, “DS2 Expressions,” in *SAS DS2 Language Reference*

Details

The YYQ function returns a SAS date value that corresponds to the first day of the specified quarter. If either *year* or *quarter* is null or missing, or if the quarter value is not valid, the result is a null or missing value.

Example

The following statements illustrate the YYQ function:

Statements	Results
DateValue=yyq(2006,3); Date7Value = put(DateValue, date7.); Date9Value = put(DateValue, date9.);	16983 01JUL06 01JUL2006
StartOfQuarter=yyq(2006,4); StartOfQuarter9=put(StartOfQuarter,date9.);	17075 01OCT2006

See Also

- Chapter 14, “Dates and Times in DS2,” in *SAS DS2 Language Reference*

Functions:

- “QTR Function” on page 288
- “YEAR Function” in *SAS DS2 Language Reference*

Chapter 6

FedSQL Expressions and Predicates

Overview of Expressions and Predicates	321
Dictionary	322
BETWEEN Predicate	322
CASE Expression	322
COALESCE Expression	326
DISTINCT Predicate	327
EXISTS Predicate	328
IN Predicate	329
IS FALSE Predicate	330
IS MISSING Predicate	331
IS NULL Predicate	333
IS TRUE Predicate	334
IS UNKNOWN Predicate	335
LIKE Predicate	336
NULLIF Expression	338
<sql-expression>	339

Overview of Expressions and Predicates

Expressions are combinations of symbols and operators that FedSQL evaluates and then returns a single value. Expressions can be as simple as a single constant or column or as complex as multiple expressions joined by an operator.

This chapter focuses on three conditional expressions, CASE, COALESCE, and NULLIF, and on <sql-expression>, a generic expression that defines all other types of expressions.

Predicates specify conditions that evaluate to either true, false, or unknown. They are used most often in WHERE and HAVING clauses and in the FROM clause in join conditions.

Dictionary

BETWEEN Predicate

Selects rows where column values are within a range of values.

Syntax

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

Arguments

expression

specifies any valid SQL expression.

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

Details

The BETWEEN predicate specifies a range of column values to select using these criteria:

- The SQL expressions must be of compatible data types.
- Because a BETWEEN condition evaluates the boundary values as a range, it is not necessary to specify the smaller quantity first.
- You can use the NOT logical operator to exclude a range of numbers. For example, you can use NOT to eliminate customer numbers between 1 and 15 (inclusive) so that you can retrieve data on more recently acquired customers.

Example

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

See Also

Expressions:

- [“<sql-expression>” on page 339](#)

CASE Expression

Selects result values that satisfy search conditions and value comparisons.

Syntax

```

CASE [case-expression]
  WHEN when-expression THEN result-expression
  ...
  [WHEN when-expression THEN result-expression]
  [ELSE result-expression]
END

```

Arguments

case-expression

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

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

when-expression

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

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

See [“<sql-expression>” on page 339](#)

result-expression

specifies an SQL expression that evaluates to a value.

See [“<sql-expression>” on page 339](#)

Details

The CASE expression selects values if certain conditions are met. The *case-expression* argument returns a single value that is conditionally evaluated for each row of a table. Use the WHEN-THEN clauses to execute a CASE expression for some, but not all of the rows in the table that is being queried or created. The optional ELSE expression gives an alternative action if no THEN expression is executed.

When you omit *case-expression*, *when-expression* is evaluated as a Boolean (true or false) value. If *when-expression* returns a nonzero, non-null result, then the WHEN clause is true. If *case-expression* is specified, then it is compared with *when-expression* for equality. If *case-expression* equals *when-expression*, then the WHEN clause is true.

If the *when-expression* is true for the row that is being executed, then the *result-expression* that follows THEN is executed. If *when-expression* is false, then FedSQL evaluates the next *when-expression* until they are all evaluated. If every *when-expression* is false, then FedSQL executes the ELSE expression, and its result becomes the CASE expression's result. If no ELSE expression is present and every *when-expression* is false, then the result of the CASE expression is null.

You can use a CASE expression as an item in the SELECT clause and as either operand in an SQL expression.

Comparisons

The COALESCE expression and the NULLIF expression are variations of the CASE expression.

The following CASE expression and COALESCE expression are equivalent:

```
case
  when value1 is not null
    then value1
  when value2 is not null
    then value2
  else value3
end

coalesce(value1, value2, value3)
```

The following CASE expression and NULLIF expression are equivalent:

```
case
  when value1 = -1 then null
  else value1
end

nullif(value1, -1);
```

Examples

Example 1: The CASE Expression Using A Search Condition

Table: [WORLDTEMPS on page 542](#)

```
select AvgLow,
  case
    when AvgLow < 32 then AvgLow + 2
    when ((AvgLow < 60) and (AvgLow > 32)) then AvgLow + 5
    when AvgLow > 60 then AvgLow + 10
    else AvgLow
  end
as Adjusted from worldtemps;
```

SAS creates the follow table:

Output 6.1 CASE Using a Search Condition

avglow	ADJUSTED
45	50
33	38
17	19
56	61
57	62
28	30
51	56
75	85
36	41
33	38
25	27

Example 2: The CASE Expression Using a ValueTable: [WORLDTEMPS on page 542](#)

```

select Country,
  case Country
    when 'Algeria' then 'Africa'
    when 'Nigeria' then 'Africa'
    when 'Netherlands' then 'Europe'
    when 'Spain' then 'Europe'
    when 'Switzerland' then 'Europe'
    when 'China' then 'Asia'
    when 'India' then 'Asia'
    when 'Venezuela' then 'South America'
    else 'Unknown'
  end
as Continent from worldtemps;

```

SAS creates the following table:

Output 6.2 CASE Using a Value

country	CONTINENT
Algeria	Africa
Netherlands	Europe
China	Asia
India	Asia
Venezuela	South America
Switzerland	Europe
China	Asia
Nigeria	Africa
Spain	Europe
China	Asia
Switzerland	Europe

See Also

Expressions:

- [“COALESCE Expression” on page 326](#)
- [“NULLIF Expression” on page 338](#)
- [<search-condition> in the “SELECT Statement” on page 386](#)

COALESCE Expression

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

Restriction: SAS data sets and SPD Engine data sets process null values as a blank string.

Syntax

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

Arguments

expression

specifies any valid SQL expression.

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

Details

COALESCE accepts one or more SQL expressions of the same data type. The COALESCE expression checks the value of each SQL expression in the order in which it is listed and returns the first non-null value. If only one SQL expression is listed, the COALESCE expression returns the value of that SQL expression. If all the values of all arguments are null, the COALESCE expression returns a null value.

In some SQL DBMSs, the COALESCE expression is called the IFNULL expression.

Note: If your query contains a large number of COALESCE expressions, it might be more efficient to use a natural join instead. For more information, see [“Natural Joins” on page 399](#).

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 322](#)

DISTINCT Predicate

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

Syntax

- Form 1: *function* **DISTINCT** (*expression*);
- Form 2: *row-value-constructor* **IS DISTINCT FROM** *row-value-constructor*
- Form 3: **SELECT DISTINCT** <select-list> FROM <table-expression>;

Arguments

function

can be any aggregate function.

expression

specifies any valid SQL expression.

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

row-value constructor

specifies any row value expression that represents a row value.

See [“Row Value Expressions” on page 51](#)

SELECT <select-list> FROM <table-expression>

is a query that retrieves rows from a table.

See For more information about using the DISTINCT predicate in the SELECT statement, see the [“SELECT Clause” on page 389](#).

Details

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

Note: Two null values are *not* considered distinct.

Example

- `select count(distinct avghigh) from worldtemps;`
- `(42, 47,15) is distinct from (42, 47, 15)`
- `select distinct c1.employee, firstname, salary
from company as c1;`

See Also**Statements:**

- [“SELECT Statement” on page 386](#)

EXISTS Predicate

Tests whether a subquery returns one or more rows.

Syntax

[NOT] EXISTS (*select-statement*)

Arguments**select-statement**

specifies a subquery with the SELECT statement.

See [“SELECT Statement” on page 386](#)

Details

The EXISTS predicate is an operator whose right operand is a subquery. The result of an EXISTS predicate is true if the subquery resolves to at least one row. The result of a NOT EXISTS predicate is true if the subquery evaluates to zero rows.

Example

The following query subsets PAYROLL based on the criteria in the subquery. If the value for STAFF.IDNUM is on the same row as the value CT in STAFF, then the matching IDNUM in PAYROLL is included in the output. Thus, the query returns all the employees from PAYROLL who live in CT.

```
select *
  from payroll p
    where exists (select * from staff s
                where p.idnumber=s.idnum and state='CT');
```

See Also

Statements:

- [“SELECT Statement” on page 386](#)

IN Predicate

Tests set membership.

Syntax

expression [NOT] IN (*select-statement* | *constant* [, ...*constant*])

Arguments

expression

specifies any valid SQL expression.

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

select-statement

specifies a subquery with the SELECT statement.

See [“SELECT Statement” on page 386](#)

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 542](#)

```
select city, country
  from worldtemps
 where avghigh in (90, 97);
```

SAS creates the following table:

Output 6.3 IN Predicate Example Output Table

city	country
Algiers	Algeria
Calcutta	India
Lagos	Nigeria

IS FALSE Predicate

Tests for a false value.

Syntax

(expression) IS [NOT] FALSE

Arguments

expression

specifies any valid SQL expression.

See “<sql-expression>” on page 339

“FedSQL Expressions” on page 42

Details

IS FALSE is a predicate that tests for a false value. IS FALSE is used in the WHERE, ON, and HAVING clauses. The IS FALSE predicate resolves to true if the result of the SQL expression is false and resolves to false if it is true.

Comparisons

The IS TRUE predicate tests for true values.

Example

Table: [WORLDCITYCOORDS](#) on page 542

```
select city
  from worldcitycoords
  where (latitude = 40) is false;
```

SAS creates the following table:

Output 6.4 IS FALSE Example Output Table

city
Algiers
Shanghai
Hong Kong
Bombay
Calcutta
Amsterdam
Lagos
Zurich
Caracas

See Also

Predicates:

- “IS TRUE Predicate” on page 334
- “IS UNKNOWN Predicate” on page 335
- <search-condition> in the “SELECT Statement” on page 386

IS MISSING Predicate

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

Syntax

expression IS [NOT] MISSING

Arguments

expression

specifies any valid SQL expression.

See “<sql-expression>” on page 339

“FedSQL Expressions” on page 42

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 542

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

SAS creates the following table:

Output 6.5 IS MISSING Example Output Table

city	country	latitude	longitude
	China	40	116

See Also

Predicates:

- “IS NULL Predicate” on page 333
- <search-condition> in the “SELECT Statement” on page 386

IS NULL Predicate

Tests for a null value.

Syntax

expression IS [NOT] NULL

Arguments

expression

specifies any valid SQL expression.

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

Details

IS NULL is a predicate that tests for a null value. IS NULL is used in the WHERE, ON, and HAVING clauses. The IS NULL predicate resolves to true if the result of the SQL expression is null and resolves to false if it is not null.

Comparisons

The IS MISSING predicate tests for SAS missing values in SAS native data stores.

Example

Table: [WORLDCITYCOORDS on page 542](#)

```
select city
  from worldcitycoords
  where latitude is not null;
```

SAS creates the following table:

Output 6.6 IS NULL Example Output Table

city
Algiers
Shanghai
Hong Kong
Bombay
Calcutta
Amsterdam
Lagos
Madrid
Zurich
Caracas

See Also

Predicates:

- “IS MISSING Predicate” on page 331
- <search-condition> in the “SELECT Statement” on page 386

IS TRUE Predicate

Tests for a true value.

Syntax

(expression) IS [NOT] TRUE

Arguments

expression

specifies any valid SQL expression.

See “<sql-expression>” on page 339

[“FedSQL Expressions” on page 42](#)

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 542](#)

```
select city
  from worldcitycoords
 where (latitude = 40) is true;
```

SAS creates the following table:

Output 6.7 IS TRUE Example Output

city
Madrid

See Also

Predicates:

- [“IS FALSE Predicate” on page 330](#)
- [“IS UNKNOWN Predicate” on page 335](#)

IS UNKNOWN Predicate

Tests for an unknown value.

Syntax

expression IS [NOT] UNKNOWN

Arguments

expression
specifies any valid SQL expression.

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

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 330](#)
- [“IS TRUE Predicate” on page 334](#)
- [<search-condition> in the “SELECT Statement” on page 386](#)

LIKE Predicate

Tests for a matching pattern.

Syntax

expression [NOT] **LIKE** *expression*

Arguments

expression

specifies any valid SQL expression that is either a character string type or a binary string type.

Tip The SQL expression on the right side of the syntax, that is the pattern, is most likely to be a literal.

See [“<sql-expression>” on page 339](#)

[“FedSQL Expressions” on page 42](#)

Details

Overview of the LIKE Predicate

The LIKE predicate selects rows by comparing character strings with a pattern-matching specification. It resolves to true and displays the matched string or strings if the left operand matches the pattern that is specified by the right operand.

Escape characters are not supported.

Note: If no rows are returned, the result is a null value.

Patterns for Searching

Patterns include three classes of characters:

underscore (`_`)

matches any single character.

percent sign (`%`)

matches any sequence of zero or more characters.

any other character

matches that character.

These patterns can appear before, after, or on both sides of characters that you want to match. The LIKE condition is case-sensitive.

The following list uses these values: **Smith**, **Smooth**, **Smothers**, **Smart**, and **Smuggle**.

`'Sm%'`

matches **Smith**, **Smooth**, **Smothers**, **Smart**, **Smuggle**.

`'%th'`

matches **Smith**, **Smooth**.

`'S__gg%'`

matches **Smuggle**.

`'S_o'`

matches a three-letter word, so it has no matches here.

`'S_o%'`

matches **Smooth**, **Smothers**.

`'S%th'`

matches **Smith**, **Smooth**.

`'M'`

matches the single, uppercase character **m** only, so it has no matches here.

Searching for Mixed-Case Strings

To search for mixed-case strings, use the UPPER function to make all the names uppercase before entering the LIKE condition:

```
upper(name) like 'SM%';
```

Note: When you are using the % character, be aware of the effect of trailing blanks. You might have to use the TRIM function to remove trailing blanks in order to match values.

Example

Table: [DENSITIES](#) on page 539

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

See Also

Functions:

- [“TRIM Function”](#) on page 313

NULLIF Expression

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

Restriction: The BASE file format processes a null value as DOUBLE values in some situations and as a blank string in other situations. For more information, see [“How FedSQL Processes Nulls and SAS Missing Values” on page 18](#).

Syntax

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

Arguments

expression

specifies any valid SQL expression.

Data type All data types are valid.

See [“<sql-expression>” on page 339](#)
[“FedSQL Expressions” on page 42](#)

Details

The NULLIF expression compares two SQL expressions and, if they are equal, returns a null value. The NULLIF expression enables you to replace a missing or inapplicable value with a null value and to use SQL's behavior for null values.

Comparisons

The NULLIF expression is a shorthand syntax for a special CASE expression. For example, if a student misses a test, a -1 is entered in the GRADES table. To replace this -1 with a null value, you could use the following CASE code.

```
update grades
  set testscore =
    CASE
      when testscore = '-1' then null
    ELSE testscore
    END;
```

The following code uses the shorter NULLIF expression.

```
update grades
  set testscore = NULLIF(testscore, '-1');
```

The IFNULL function compares two SQL expressions and returns the second SQL expression if the first SQL expression is a null value. The NULLIF expression compares two SQL expressions and returns a null value if the two SQL expressions are equal.

Example

Table: [WORLD_CITYCOORDS on page 542](#)

```

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

```

See Also

Expressions:

- [“CASE Expression” on page 322](#)
- [“COALESCE Expression” on page 326](#)

Functions:

- [“IFNULL Function” on page 246](#)

<sql-expression>

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

Syntax

```

<sql-expression> ::=
    constant
  | variable
  | [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.

variable

is the name of a user-defined variable.

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 5, “FedSQL Functions,” on page 177](#)

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 322](#) or [“COALESCE Expression” on page 326](#).

Table 6.1 Valid Operators

Operator	Description
+	adds
–	subtracts
*	multiplies
/	divides
=	equals
≠	does not equal
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to
**	raises to a power
unary –	indicates a negative number
	concatenates

predicate

is an expression that returns true, false, or unknown.

Valid predicates are as follows.

- [“BETWEEN Predicate” on page 322](#)
- [“DISTINCT Predicate” on page 327](#)
- [“EXISTS Predicate” on page 328](#)
- [“IN Predicate” on page 329](#)
- [“IS FALSE Predicate” on page 330](#)
- [“IS MISSING Predicate” on page 331](#)
- [“IS NULL Predicate” on page 333](#)

- [“IS TRUE Predicate” on page 334](#)
- [“IS UNKNOWN Predicate” on page 335](#)
- [“LIKE Predicate” on page 336](#)

Details

Overview of <sql-expression>

Simple expressions can be a single constant, variable, column name, or function. Complex expressions are two or more simple expressions that are joined by an operator or predicate.

Functions in Expressions

An expression can contain a SAS function or an aggregate function. SAS functions perform a computation or system manipulation on one or more arguments and return a value. Aggregate functions produce a statistical summary of data in the entire table that is listed in the FROM clause or for each group that is specified in a GROUP BY clause. If GROUP BY is omitted, then all the rows in the table are considered to be a single group. Aggregate functions reduce all the values in each row or column in a table to one summarizing or aggregate value. For example, the sum (one value) of a column results from the addition of all the values in the column.

Subqueries in Expressions

FedSQL allows a scalar subquery (contained in parentheses) at any point in an expression where a simple column value or constant can be used. In this case, a subquery must return a single value (that is, one row with only one column).

Order of Evaluation

The operators and predicates that are shown in the following table are listed in the order in which they are evaluated.

Table 6.2 Expressions, Operators, and Predicates and Order of Evaluation

Group	Expressions, Operators, and Predicates	Description
0	()	forces the expression enclosed to be evaluated first
1	CASE expression	See “CASE Expression” on page 322
2	**	raises to a power
	unary +, unary –	indicates a positive or negative number
3	*	multiplies
	/	divides
4	+	adds
	–	subtracts

Group	Expressions, Operators, and Predicates	Description
5		concatenates
6	[NOT] BETWEEN predicate	See “ BETWEEN Predicate ” on page 322
	DISTINCT predicate	See “ DISTINCT Predicate ” on page 327
	[NOT] EXISTS predicate	See “ EXISTS Predicate ” on page 328
	[NOT] IN predicate	See “ IN Predicate ” on page 329
	IS [NOT] TRUE predicate	See “ IS TRUE Predicate ” on page 334
	IS [NOT] FALSE predicate	See “ IS FALSE Predicate ” on page 330
	IS [NOT] MISSING predicate	See “ IS MISSING Predicate ” on page 331
	IS [NOT] NULL predicate	See “ IS NULL Predicate ” on page 333
	IS [NOT] UNKNOWN predicate	See “ IS UNKNOWN Predicate ” on page 335
	LIKE predicate	See “ LIKE Predicate ” on page 336
7	=	equals
	\neq , \neq	does not equal
	>	is greater than
	<	is less than
	>=	is greater than or equal to
	<=	is less than or equal to
8	AND	indicates logical AND
9	OR	indicates logical OR
10	NOT	indicates logical NOT

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

You can use parentheses to group values or to nest mathematical expressions. Parentheses make expressions easier to read and can also be used to change the order of evaluation of the operators. Evaluating expressions with parentheses begins at the deepest level of parentheses and moves outward. For example, SAS evaluates $A+B*C$ as

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

See Also

Statements:

- [“SELECT Statement” on page 386](#)
- [“FedSQL Expressions” on page 42](#)

Chapter 7

FedSQL Informat

Definition of an Informat	345
General Informat Syntax	345
How Informat Are Used in FedSQL	346
How to Specify Informat in FedSQL	347
Validation of FedSQL Informat	347
FedSQL Informat Example	347

Definition of an Informat

An *informat* is an instruction that determines how values are read into a column. For example, the following value contains a dollar sign and commas:

```
$1,000,000
```

To remove the dollar sign (\$) and commas (,) before storing the numeric value 1000000 in a variable, read this value with the COMMA11. informat.

General Informat Syntax

FedSQL inmat have the following syntax:

```
[$]informat[w].[d]
```

Arguments

\$

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

informat

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

w

specifies the informat width, which for most inmat is the number of columns in the input data.

d

specifies an optional decimal scaling factor in the numeric informats. SAS divides the input data by 10 to the power of *d*.

Note: Even though SAS can read up to 31 decimal places when you specify some numeric informats, floating-point numbers with more than 12 decimal places might lose precision because of the limitations of the eight-byte, floating-point representation used by most computers.

Informats always contain a period (.) as a part of the name. If you omit the *w* and the *d* values from the informat, SAS uses default values. If the data contains decimal points, SAS ignores the *d* value and reads the number of decimal places that are actually in the input data.

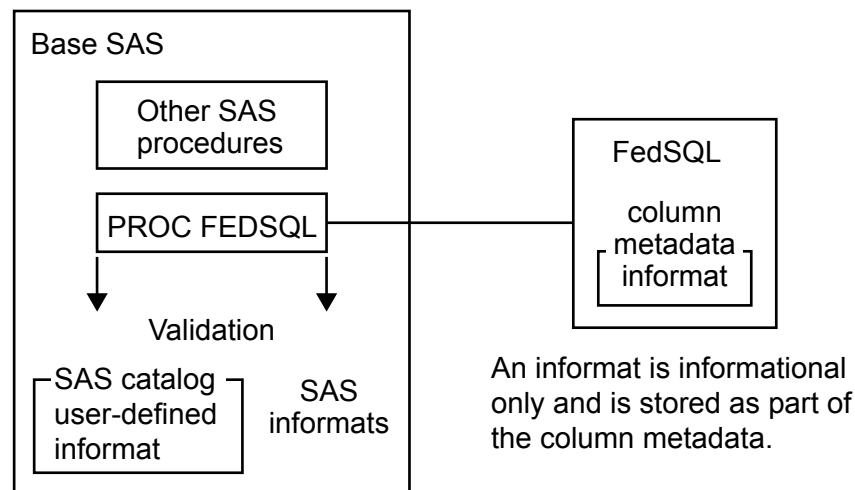
For more information about how informats work and a complete list of informats, see *SAS Formats and Informats: Reference*.

How Informats Are Used in FedSQL

Informats are informational and do not operate on your data at run time in the FedSQL environment. The client application is responsible for using the informat to convert the raw data when the table is returned to the application.

Note: If the informat that is stored in the FedSQL environment is invalid, an error occurs only when the invalid informat is used in the client application.

Figure 7.1 Diagram of How Informats Work



An informat is applied by the client application. Here, the Base SAS session validates and applies the informat when a SAS procedure uses the data.

However, in FedSQL, you can store and retrieve informat names. The informat name is associated with a column—either temporarily for the duration of an operation or permanently—by storing the informat as a metadata attribute on the column. The metadata then can be retrieved for subsequent operations.

FedSQL supports SAS informats as follows.

- Both informats supplied by SAS and user-defined informats can be associated with a column. For information about how to create your own informat in SAS, see PROC FORMAT in *Base SAS Procedures Guide*.

Note: To create and access user-defined informats, a Base SAS session must be available in order to access the SAS catalog file that stores the SAS informat definitions.

- Only the SAS data sets and SPD Engine data sets support storing and retrieving an informat with a column.
- Informats can be associated with all data types, but all data types are converted to either CHAR or DOUBLE.
- You can associate SAS informats with a column by using the HAVING clause of the FedSQL CREATE TABLE or the SET clause of the FedSQL ALTER TABLE statement. For more information, see [“How to Specify Informats in FedSQL” on page 347](#).

For more information and a complete list of informats that are supplied by SAS, see the section on informats in *SAS Formats and Informats: Reference*.

How to Specify Informats in FedSQL

In FedSQL, specify informats as an attribute in the HAVING clause of the CREATE TABLE statement. For example, in the following statement, the column X is declared with the IEEE8.2 format and the BITS5.2 informat.

```
create table a (x double having format ieee8.2 informat bits5.2 label 'foo');
```

Validation of FedSQL Informats

Informats are not validated by a data source or applied to a column until execution time.

When a table is created using FedSQL, no validation occurs.

When metadata for a column is requested, the informat name is returned without validation.

FedSQL Informat Example

```
create table y (x double having label 'price' format $5. informat $charzb4.3);
```


Chapter 8

FedSQL Statements

Overview of Statements	349
FedSQL Statements by Category	350
Dictionary	351
ALTER TABLE Statement	351
BEGIN Statement	355
COMMIT Statement	356
CREATE INDEX Statement	357
CREATE TABLE Statement	359
CREATE VIEW Statement	375
DESCRIBE VIEW Statement	376
DELETE Statement	377
DROP INDEX Statement	378
DROP TABLE Statement	379
DROP VIEW Statement	380
EXECUTE Statement	381
INSERT Statement	382
ROLLBACK Statement	385
SELECT Statement	386
UPDATE Statement	412

Overview of Statements

A FedSQL statement is a series of items that can include keywords, identifiers, special characters, and operators. All FedSQL statements end with a semicolon.

There are three categories of statements:

data definition

statements that are used to create, modify, and delete tables and views.

data control

statements that are used to control transactions with the database.

data manipulation

statements that are used to add, update, and delete data.

SQL pass-through facility

statements that connect to a DBMS and enable you to send DBMS-specific SQL statements directly to the DBMS.

FedSQL Statements by Category

Category	Language Elements	Description
Data Control	BEGIN Statement (p. 355)	Marks the beginning of a transaction that comprises multiple statements.
	COMMIT Statement (p. 356)	Makes changes that have been performed since the start of a transaction a permanent part of the database.
	ROLLBACK Statement (p. 385)	Rolls back transaction changes to the beginning of the transaction.
Data Definition	ALTER TABLE Statement (p. 351)	Adds or drops table columns and modifies column definitions.
	CREATE INDEX Statement (p. 357)	Creates an index on columns in a specified table.
	CREATE TABLE Statement (p. 359)	Creates a new table.
	CREATE VIEW Statement (p. 375)	Creates a view of data from one or more tables or other views.
	DESCRIBE VIEW Statement (p. 376)	Retrieves SQL from a view and returns a result set.
	DROP INDEX Statement (p. 378)	Removes the specified index from a table.
	DROP TABLE Statement (p. 379)	Removes a table from the database.
	DROP VIEW Statement (p. 380)	Removes a view from the database.
Data Manipulation	SELECT Statement (p. 386)	Retrieves columns and rows of data from tables.
	DELETE Statement (p. 377)	Deletes rows from a table.
	INSERT Statement (p. 382)	Adds rows to a specified table.
	UPDATE Statement (p. 412)	Modifies a column's values in existing rows of a table.
FedSQL Pass-through	EXECUTE Statement (p. 381)	Sends a DBMS-specific SQL statement to a DBMS that FedSQL supports.

Dictionary

ALTER TABLE Statement

Adds or drops table columns and modifies column definitions.

- Category:** Data Definition
- Restriction:** Constraint syntax is defined by the data source. The constraint syntax that is provided here is a general syntax. For complete constraint syntax, see the documentation for your data source. The SPD Engine data source does not support constraints.
- Supports:** [“EXECUTE Statement” on page 381](#)
- Data source:** SAS data set, Aster, DB2 under UNIX and PC, Greenplum, Hive, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata
- 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

```
ALTER TABLE table [\{OPTIONS SAS-table-option=value [ ...SAS-table-option=value] \}]
  ADD COLUMN <column-definition> [, ...<column-definition>]
  | ADD CONSTRAINT <table-constraint>
  | ALTER [COLUMN] <column-definition> [, ...<column-definition> ]
  | DROP COLUMN column [, ...column] [FORCE]
  | DROP CONSTRAINT constraint [, ...constraint]
  | RENAME TO table
  | RENAME column TO column
;

<column-definition>::=
  column data-type [<column-constraint>] [ SET DEFAULT value | DROP DEFAULT ]

<column-constraint>::=
  CONSTRAINT constraint
  { CHECK (search-condition)
  | PRIMARY KEY
  | UNIQUE
  | NOT NULL }
```

```

<table-constraint>::=
    CONSTRAINT constraint
    { CHECK (search-condition)
    | PRIMARY KEY (column [, ...column])
    | UNIQUE (column [, ...column])
    | <referential-constraint> }
    [<constraint-check-time>]

<referential-constraint>::=
    FOREIGN KEY (referencing-column [, ... referencing-column])
        REFERENCES referenced-table (referenced-column [, ...referenced-column])
        [<referential-trigger-action>]

<referential-trigger-action>::=
    { [ON UPDATE <referential-action> [ON DELETE <referential-action>]] }
    | { [ON DELETE <referential-action> [ON UPDATE <referential-action>]] }

<referential-action>::=
    CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION

<constraint-check-time>::=
    { DEFERRABLE [INITIALLY DEFERRED | INITIALLY IMMEDIATE] }
    | { [INITIALLY DEFERRED | INITIALLY IMMEDIATE] DEFERRABLE }
    | INITIALLY DEFERRED
    | { NOT DEFERRABLE [INITIALLY IMMEDIATE] }
    | { [INITIALLY IMMEDIATE] NOT DEFERRABLE }

```

Arguments

table

specifies a table to modify. *table* can be specified in one of these forms:

- *catalog.schema.table-name*
- *schema.table-name*
- *catalog.table-name*
- *table-name*

catalog

is an implementation of the ANSI SQL standard for an SQL catalog. The catalog is a data container object that groups logically related schemas. It is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.

schema

is an implementation of the ANSI SQL standard for an SQL schema. The schema is a data container object that groups files such as tables and views and other objects that are supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

table-name

is the name of the table.

Restriction You cannot alter an MDS table while it is referenced in another transaction or statement. If a request fails, make sure other users are no longer using the table or have disconnected.

Requirement Table naming conventions are based on the data source. When more than one data source is involved, the maximum length of a table name is determined by the maximum length that is supported by all of the data sources and FedSQL. For example, if your data sources are a SAS data set that has a maximum of 32 characters and MySQL that has a maximum of 64 characters, the maximum length of a table name is 32 characters. For more information, see the documentation for your data source.

{OPTIONS *SAS-table-option*=value [, ...*SAS-table-option*=value]}

specifies one or more table options and their respective values to apply to the table.

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

See [Chapter 9, “FedSQL Statement Table Options,” on page 415](#)

column

specifies the name of a column in a table.

Requirement Each column in a table must be unique.

ADD COLUMN <column-definition> [, ...<column-definition>]

specifies to add a column to a table.

Data source Aster, DB2 under UNIX and PC, Greenplum, Hive, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

See [column on page 353](#) and “<column-definition> Arguments” on page 362 in the CREATE TABLE statement

ADD CONSTRAINT <table-constraint>

specifies to add an integrity constraint to one or more columns.

Data source SAS data set, Aster, DB2 under UNIX and PC, Greenplum, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

Note Constraint validation and enforcement is performed by the data source. You should be familiar with data source’s requirements before adding or dropping an integrity constraint.

See “<column-constraint> and <table-constraint> Arguments” on page 369 in the CREATE TABLE statement

ALTER [COLUMN] <column-definition> [, ...<column-definition>]

specifies to modify the definition of one or more columns.

Data source Aster, DB2 under UNIX and PC, Greenplum, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, Sybase IQ, Teradata

See “<column-definition> Arguments” on page 362 in the CREATE TABLE statement

DROP COLUMN *column* [, ...*column*] [FORCE]

specifies to delete the specified column from the table. When FORCE is specified, the column is dropped from the table without error processing. Use the FORCE keyword only when you are certain that dropping the column without error processing will not negatively affect the table.

Data source Aster, DB2 under UNIX and PC, Greenplum, Hive, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

See [“*column*” on page 353](#)

DROP CONSTRAINT *constraint* [, ...*constraint*]

specifies to delete an integrity constraint.

constraint

specifies the name of the constraint to drop.

Data source SAS data set, Aster, DB2 under UNIX and PC, Greenplum, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

RENAME

changes the specified table or column name to a new name. The new name follows the TO keyword.

Data source Aster, DB2 under UNIX and PC, Greenplum, MDS, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

<column-definition> Arguments

See [“<column-definition> Arguments” on page 362](#) in the CREATE TABLE statement.

<column-constraint> and <table-constraint> Arguments

See [“<column-constraint> and <table-constraint> Arguments” on page 369](#) in the CREATE TABLE statement.

Details***Adding and Deleting Columns***

To add a column to a table, use the ALTER TABLE statement with the ADD COLUMN clause. By using the ADD COLUMN clause, you name the column and define the column attributes, such as the column data type, a default value, and a label for the column.

```
alter table personal_info add column em_num char(16);
```

To delete a column, you need to specify only the column name.

```
alter table personal_info drop em_num;
```

Adding and Deleting Constraints

You can define a constraint on a column that is added with the ADD COLUMN clause in all data sources that support integrity constraints as follows:

```
alter table mytest add y char(5) constraint c2 unique;
```

An exception is SAP HANA. For SAP HANA, you must add the column first, then add the constraint with ADD CONSTRAINT.

By using the ADD CONSTRAINT clause, you can add a constraint for a table, or you can add or modify a constraint for one or more existing columns. The following code adds a primary key constraint to a table with ALTER TABLE.

```
alter table customers add constraint pkey primary key(custid);
```

The following code adds a CHECK constraint to an existing column.

```
alter table mytest add constraint chk check (col1 < 5);
```

You delete a constraint by using the DROP CONSTRAINT clause. The following code is an example of deleting a constraint from a SAS data set. The code deletes a primary key constraint that was defined at table creation. When they are defined at table creation, primary keys in a SAS data set are created with the default name `_pk0001_`.

```
alter table customers drop constraint _pk0001_;
```

When you add or modify constraints, you must know the constraint syntax that the data source supports. Constraint validation is performed by the data source.

Altering Column Definitions

To alter a column definition, use the ALTER TABLE statement with the ALTER clause.

```
alter table personal_info alter country set default 'UK';
```

This example drops a default value from a column.

```
alter table personal_info alter country drop default;
```

This example renames a column.

```
alter table sales rename column y to customers;
```

Comparisons

You use the ALTER TABLE statement to alter a table after it has been created. Use the CREATE TABLE statement to create a table.

See Also

- [“How to Store, Change, Delete, and Use Stored Formats” on page 70](#)

Statements:

- [“CREATE TABLE Statement” on page 359](#)
- [“DROP INDEX Statement” on page 378](#)

BEGIN Statement

Marks the beginning of a transaction that comprises multiple statements.

Category: Data Control

Restriction: The BEGIN statement has an effect only when autocommit functionality is off.

Supports: [“EXECUTE Statement” on page 381](#)

Data source: Greenplum, MDS, MySQL, ODBC, Oracle, SAP HANA, Sybase IQ, Teradata

Syntax

BEGIN [TRANSACTION];

Details

The BEGIN statement marks a point at which the data is logically and physically consistent. If errors are encountered during a transaction, the user can roll back all changes to the data to this last known state of consistency.

When you use FedSQL processing, this statement is not really necessary. By default, autocommit functionality is on, which means that all updates are committed immediately after each request is submitted, and no rollback is possible. A transaction is effectively started with each update. The BEGIN statement is provided to enable you to mark the beginning of a transaction that comprises multiple statements.

See the server administration documentation for information about how to turn off autocommit functionality. For example, see *SAS Federation Server: Administrator's Guide* for the appropriate connection option to the FedSQL driver. For the FEDSQL procedure, see *Base SAS Procedures Guide*.

Note: BEGIN has no effect on the SASHDAT data source.

See Also

Statements:

- [“COMMIT Statement” on page 356](#)
- [“ROLLBACK Statement” on page 385](#)

COMMIT Statement

Makes changes that have been performed since the start of a transaction a permanent part of the database.

Category:	Data Control
Restriction:	The COMMIT statement has an effect only when autocommit functionality is off.
Supports:	“EXECUTE Statement” on page 381
Data source:	SASHDAT, Greenplum, MDS, MySQL, ODBC, Oracle, SAP HANA, Sybase IQ, Teradata

Syntax

COMMIT [TRANSACTION];

Details

When your program has completed all of the statements in the transaction, you must explicitly terminate the transaction using COMMIT or ROLLBACK. You use a COMMIT statement to make the changes to the database permanent.

You cannot roll back the changes to the database after the COMMIT statement is executed.

Note: The COMMIT statement has an effect only when autocommit functionality is off. In most data sources, autocommit functionality is on by default. Refer to the server administration documentation for information about how to turn off autocommit functionality. For example, see the *SAS Federation Server: Administrator's Guide* for the appropriate connection option to the FedSQL driver. For the FEDSQL procedure, see the *Base SAS Procedures Guide*.

Note: For the SASHDAT data source, the COMMIT statement does not permanently alter the database. Instead, it closes the SASHDAT table, either when a statement is unprepared or when the connection is disconnected (default), depending on the setting of the COMMIT= connection option. If a SASHDAT table remains open when autocommit functionality is on, COMMIT can be used to close it.

Comparisons

The ROLLBACK statement causes all the uncommitted changes that were made by the transaction to be rolled back to the start of the transaction or to the point that is marked by the BEGIN statement. The COMMIT statement takes all the data changes that have been performed since the start of the transaction and makes them a permanent part of the database.

See Also

Statements:

- [“BEGIN Statement” on page 355](#)
- [“ROLLBACK Statement” on page 385](#)

CREATE INDEX Statement

Creates an index on columns in a specified table.

Category: Data Definition

Supports: [“EXECUTE Statement” on page 381](#)

Data source: SAS data set, SPD Engine data set, Aster, DB2 under UNIX and PC, Greenplum, MDS, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

Syntax

```
CREATE [UNIQUE] INDEX index ON table
(column [ASCENDING | DESCENDING ][, ...column]);
```

Arguments

UNIQUE

creates a unique index on a table.

index

specifies the name of the index.

Restriction *For SAS data sets* – If you are creating an index on one column only, then *index* must be the same as *column*. If you are creating an index on

more than one column, then *index* cannot be the same as any column in the table.

table

specifies the name of the table that contains the column or columns to be indexed.

column

specifies the name of the column to which the index applies. Specify two or more column names to create a composite index.

Tip If you search two or more columns as a unit or if you have queries that involve only specific columns, use composite indexes.

ASCENDING

Rows are sorted from the smallest value to the largest value. This is the default value.

Alias ASC

DESCENDING

Observations are sorted from the largest value to the smallest value.

Alias DESC

Details

Overview of Table Indexes

An *index* stores both the values of a table's columns and a system of directions that enable access to rows in that table by index value. Defining an index on a column or set of columns enables SAS, under certain circumstances, to locate rows in a table more quickly and efficiently. Indexes enable SAS to execute the following classes of queries more efficiently:

- comparisons against a column that is indexed
- an IN subquery where the column in the inner subquery is indexed
- correlated subqueries, where the column being compared with the correlated reference is indexed
- join-queries, where the join-expression is an equals comparison and all the columns in the join-expression are indexed in one of the tables being joined

SAS maintains indexes for all changes to the table, whether the changes originate from FedSQL or from some other source. Therefore, if you alter a column's definition or update its values, then the same index continues to be defined for it. However, if an indexed column in a table is dropped, then the index on it is also dropped.

You can create simple or composite indexes. A *simple index* is created on one column in a table. A simple index must have the same name as that column. A *composite index* is one index name that is defined for two or more columns. The columns can be specified in any order, and they can have different data types. A composite index name cannot match the name of any column in the table. If you drop a composite index, then the index is dropped for all the columns that are named in that composite index.

Only the table owner can create an index on a table.

When you create an index, you must know the syntax that the data source supports. Index validation is performed by the data source. FedSQL provides the

DBCREATE_INDEX_OPTS= table option to enable you to specify data source-specific index parameters for ODBC databases.

UNIQUE Keyword

The UNIQUE keyword causes SAS to reject any change to a table that would cause more than one row to have the same index value. Unique indexes guarantee that data in one column, or in a composite group of columns, remain unique for every row in a table. A unique index can be defined for a column that includes null or SAS missing values if each row has a unique index value.

See Also

Statements:

- [“DROP INDEX Statement” on page 378](#)

Table Options:

- [“DBCREATE_INDEX_OPTS=” on page 450](#)

CREATE TABLE Statement

Creates a new table.

Category:	Data Definition
Restriction:	Constraint syntax is defined by the data source. The constraint syntax provided here is a general syntax. For complete constraint syntax, see the documentation for your datasources. The SPD Engine data source does not support constraints.
Supports:	“EXECUTE Statement” on page 381
Data source:	SAS data set, SPD Engine data set, SASHDAT file, Aster, DB2 under UNIX and PC, Greenplum, HDMD, Hive, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata
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 | _NULL_}
[{OPTIONS SAS-table-option=value
...SAS-table-option=value}]
| ( <column-definition> [, ...<column-definition> | <table-constraint>])
| AS query-expression
;

<column-definition>::=
column data-type [<column-constraint>] [DEFAULT value]
[HAVING [FORMAT format][INFORMAT informat][LABEL 'label']]
```

<column-constraint>::=

```

CONSTRAINT constraint
{ CHECK (search-condition)
| PRIMARY KEY
| UNIQUE
| NOT NULL }

```

<table-constraint>::=

```

CONSTRAINT constraint
{ CHECK (search-condition)
| PRIMARY KEY (column [, ...column])
| UNIQUE (column [, ...column])
| <referential-constraint> }
[<constraint-check-time>]

```

<referential-constraint>::=

```

FOREIGN KEY (referencing-column [, ...referencing-column])
REFERENCES referenced-table (referenced-column [, ...referenced-column])
[<referential-trigger-action>]

```

<referential-trigger-action>::=

```

{ ON UPDATE <referential-action> [ON DELETE <referential-action>] }
| { ON DELETE <referential-action> [ON UPDATE <referential-action>] }

```

<referential-action>::=

```

CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION

```

<constraint-check-time>::=

```

{ DEFERRABLE [INITIALLY DEFERRED | INITIALLY IMMEDIATE] }
| { [INITIALLY DEFERRED | INITIALLY IMMEDIATE] DEFERRABLE }
| INITIALLY DEFERRED
| { NOT DEFERRABLE [INITIALLY IMMEDIATE] }
| { [INITIALLY IMMEDIATE] NOT DEFERRABLE }

```

Arguments

table

specifies a table to modify. *table* can be one of these forms

- *catalog.schema.table-name*
- *schema.table-name*
- *catalog.table-name*
- *table-name*

catalog

is an implementation of the ANSI SQL standard for an SQL catalog. The catalog is a data container object that groups logically related schemas. The catalog is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.

schema

is an implementation of the ANSI SQL standard for an SQL schema. The schema is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

table-name

is the name of the table.

Requirement Table naming conventions are based on the data source. When more than one data source is involved, the maximum length of a table name is determined by the maximum length that is supported by all of the data sources and FedSQL. For example, if your data sources are a SAS data set that has a maximum of 32 characters and MySQL that has a maximum of 64 characters, the maximum length of a table name is 32 characters. For more information, see the documentation for your data source.

NULL

specifies to test the performance of creating a table using the *AS query-expression* clause. FedSQL creates the table internally as if it were to be saved, writing the normal progress messages. The table creation appears to be successful. Once the query expression is complete, the table is discarded.

Interaction A *SELECT select-list FROM _NULL_* statement returns an error.

{*OPTIONS SAS-table-option=value* [... *SAS-table-option=value*]}

specifies one or more table options and their respective values to apply to the table.

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

See [Chapter 9, “FedSQL Statement Table Options,” on page 415](#)

AS query-expression

specifies to create a new table from an existing table by selecting rows from the existing table using a query expression. The column attributes, such as formats and labels, are copied from the existing table to the new table.

query-expression

specifies the *SELECT* statement that retrieves information from an existing table to use in creating a new table.

Requirement The number of columns used in the *SELECT* statement must equal the number of columns in the table.

See [“Creating and Populating Tables from a Query Expression” on page 373](#)

[“Query Expressions and Subqueries” on page 42](#)

[“SELECT Statement” on page 386](#)

Data source SAS data set, SPD Engine data set, Aster, DB2 under UNIX and PC, Greenplum, HDMD, Hive, MDS, MySQL, Netezza, ODBC, Oracle, SAP HANA, Sybase IQ, Teradata

<column-definition> Arguments**column**

specifies the name of the column. The column name is either a local or schema qualified name, using one of these forms:

- *schema.column-name*
- *column-name*

schema

is an implementation of the ANSI SQL standard for an SQL schema. The schema is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

column-name

is the name of the column.

Restriction For SAS data sets and SPD Engine data sets, the column name cannot be longer than 32 characters.

Requirement When more than one data source is involved, the maximum length of a column name is determined by the maximum length that is supported by all of the data sources and FedSQL. For example, if your data sources are a SAS data set that has a maximum of 32 characters and MySQL that has a maximum of 64 characters, the maximum length of a column name is 32 characters. Each column in a table must be unique.

See For column name requirements, see the documentation for your data source.

data-type

specifies the type of data that the column can store. If the column definition is for a SAS data set or an SPD Engine data set, and the data type is other than CHAR or DOUBLE, FedSQL converts character data types to a CHAR and numeric data types to DOUBLE. FedSQL supports the following table data types:

BIGINT

stores a large signed, exact whole number.

Range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Data source Aster, DB2 under UNIX and PC, Greenplum, HDMD, Hive, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

Storage size 8 bytes

BINARY(*n*)

stores varying-length binary data.

n

specifies the maximum number of bytes that can be used to store the binary data. The number of bytes that are used to store the binary data is the number of bytes that are necessary to represent the binary data, up to *n* bytes.

Range

Data source DB2 under UNIX and PC, Hive, MDS, ODBC, SAP HANA, Sybase IQ

Storage size up to n bytes

CHAR(n) [CHARACTER SET "*character-set-identifier*"]
stores a fixed-length character string.

n

specifies the number of bytes that are used to store the character string. If the character string is less than n bytes, the value is right-padded with spaces.

CHARACTER SET "*character-set-identifier*"
specifies character set encoding information for CHAR data types.

Default Default encoding depends on your operating system and locale.

Restriction SAS data sets, SPD Engine data sets, and SASHDAT files do not support the use of CHARACTER SET "*character-set-identifier*". SASHDAT files use the encoding specified in the connection string. If the encoding option is not specified, the encoding defaults to the character set associated with the operating system for SAS Federation Server.

See “LOCALE= Values and Default Settings for ENCODING, PAPERSIZE, DFLANG, and DATESTYLE Options” in Chapter 20 of *SAS National Language Support (NLS): Reference Guide*

Alias CHARACTER(n)

Requirement n must be specified.

Data source SAS data set, SPD Engine data set, SASHDAT file, DB2 under UNIX and PC, Greenplum, HDMD, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Teradata

Storage size n bytes

DATE

stores a date value in the format *yyyy-mm-dd*.

Date Element	Description	Valid Values
<i>yyyy</i>	a four-digit year	0001 – 9999
<i>mm</i>	a two-digit month	01 – 12
<i>dd</i>	a two-digit day	00 – 31

Requirement Input values must be specified as a DATE constant. For more information, see [“FedSQL Date, Time, and Datetime Constants” on page 52](#).

Data source	SAS data set, SPD Engine data set, SASHDAT file, Aster, DB2 under UNIX and PC, Greenplum, HDMD, Hive, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata
--------------------	--

Storage size	8 bytes
---------------------	---------

DECIMAL

stores a signed, fixed-point decimal number.

Alias	NUMERIC
--------------	---------

Data source	Aster, DB2 under UNIX and PC, Greenplum, Hive, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata
--------------------	---

Storage size	8 bytes
---------------------	---------

DOUBLE

stores a signed, approximate, floating point number.

Alias	DOUBLE PRECISION
--------------	------------------

Data source	SAS data set, SPD Engine data set, SASHDAT file, Aster, DB2 under UNIX and PC, Greenplum, HDMD, Hive, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata
--------------------	--

Storage size	8 bytes
---------------------	---------

FLOAT(*p*)

stores a signed, approximate, single-precision or double-precision, floating point number. The user-specified precision determines whether the data type stores a single precision or double precision number. If the specified precision is equal to or greater than 25, the value is stored as a double precision number, which is a DOUBLE. If the specified precision is less than 25, the value is stored as a single precision number, which is a REAL.

p

specifies the maximum number of digits in the floating point number.

Data source	SAS data set, SPD Engine data set, Aster, DB2 under UNIX and PC, Greenplum, Hive, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata
--------------------	--

Storage size	4 bytes
---------------------	---------

Tip	If <i>p</i> is not specified, a DOUBLE is used.
------------	---

INTEGER

stores an exact whole number.

Range	-2,147,483,648 to 2,147,483,647
--------------	---------------------------------

Data source	SAS data set, SPD Engine data set, Aster, DB2 under UNIX and PC, Greenplum, HDMD, Hive, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata
--------------------	--

Storage size 4 bytes

NCHAR(*n*)

stores a fixed-length character string by using the Unicode national character set.

n

specifies the maximum number of multibyte characters that are used to store the character string. If the character string is less than *n* bytes, the value is right-padded with spaces.

Requirement *n* must be specified.

Data source Aster, DB2 under UNIX and PC, MDS, MySQL, ODBC, Oracle, SAP HANA, Sybase IQ

Storage size *n* bytes. Depending on the operating system, Unicode characters use either 2 or 4 bytes per character and support all international characters.

NUMERIC

stores a signed, fixed-point decimal number.

Alias DECIMAL

Data source Aster, DB2 under UNIX and PC, Greenplum, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, Sybase IQ, Teradata

Storage size 8 bytes

NVARCHAR(*n*)

stores a varying-length character string by using the Unicode national character set.

n

specifies the maximum number of multibyte characters that can be used to store the character string. The number of bytes that are stored is the actual number of characters specified, up to *n* bytes.

Requirement *n* must be specified.

Data source DB2 under UNIX and PC, MDS, ODBC, Oracle, Netezza, SAP HANA, Sybase IQ

Storage size up to *n* bytes. Depending on the platform, Unicode characters use either 2 or bytes per character and can support all international characters.

REAL

stores a signed, approximate, single-precision, floating-point number.

Requirement

Data source SAS data set, SPD Engine data set, Aster, DB2 under UNIX and PC, Greenplum, HDMD, Hive, MySQL, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase IQ

Storage size 4 bytes

SMALLINT

stores a small signed, exact whole number.

Range -32,768 to 32,767

Data source SAS data set, SPD Engine data set, Aster, DB2 under UNIX and PC, Greenplum, HDMD, Hive, MySQL, Netezza, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

Storage size 2 bytes

TIME(*p*)

stores a time value with seconds precision.

Time values are in the following form: *hh:mm:ss[.nnnnnn]*

Time Element	Description	Valid Values
<i>hh</i>	a two-digit hour	00 – 23
<i>mm</i>	a two-digit minute	00 – 59
<i>ss</i>	a two-digit second	00 – 61
<i>nnnnnn</i>	up to six digits to indicate a fraction of a second	0 – 999999

p specifies 0–6 digits to use for the precision of a fraction of a second.

Requirement Input values must be specified as a TIME constant. For more information, see [“FedSQL Date, Time, and Datetime Constants” on page 52](#).

Data source SAS data set, SPD Engine data set, SASHDAT file, Aster, DB2 under UNIX and PC, Greenplum, HDMD, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

Storage size 8 bytes

TIMESTAMP(*p*)

stores both the date and time value with seconds precision.

Date and time values are in the following form: *yyyy-mm-dd hh:mm:ss[.nnnnnn]*

Time Element	Description	Valid Values
Date Components		
<i>yyyy</i>	a four-digit year	0001- 9999
<i>mm</i>	a two-digit month	01- 12
<i>dd</i>	a two-digit day	01- 31

Time Element	Description	Valid Values
Time Components		
<i>hh</i>	a two-digit hour	00 – 23
<i>mm</i>	a two-digit minute	00 – 59
<i>ss</i>	a two-digit second	00 – 61
<i>nnnnnn</i>	up to six digits to indicate a fraction of a second	0 – 999999

p specifies 0–6 digits to use for the precision of a fraction of a second.

Requirement Input values must be specified as a **TIMESTAMP** constant. For more information, see [“FedSQL Date, Time, and Datetime Constants”](#) on page 52.

Data source SAS data set, SPD Engine data set, SASHDAT file, DB2 under UNIX and PC, Greenplum, HDMD, Hive, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

Storage size 8 bytes

TINYINT

stores a very small signed exact, whole number.

Range -128 to 127

Data source SAS data set, SPD Engine data set, HDMD, Hive, MySQL, Netezza, ODBC, Oracle, SAP HANA, Sybase IQ, Teradata

Storage size 1 byte

VARBINARY(*n*)

stores varying-length binary data.

n specifies the maximum number of bytes that can be used to store the binary data. The number of bytes that are used to store the binary data is the number of bytes that are necessary to represent the binary data, up to *n* bytes.

Range

Data source DB2 under UNIX and PC, MDS, ODBC, Oracle, SAP HANA, Sybase IQ, Teradata

Storage size up to *n* bytes

VARCHAR(*n*) [CHARACTER SET "*character-set-identifier*"]

stores a varying-length character string.

n

specifies the number of bytes used to store the character string. The number of bytes that are stored is the actual number of characters, up to *n* bytes.

[CHARACTER SET "*character-set-identifier*"]

specifies character set encoding information for CHAR data types.

Default Default encoding depends on your operating system and locale.

See “LOCALE= Values and Default Settings for ENCODING, PAPERSIZE, DFLANG, and DATESTYLE Options” in Chapter 20 of *SAS National Language Support (NLS): Reference Guide*

Data source Aster, DB2 under UNIX and PC, Greenplum, HDMD, Hive, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

Storage size up to *n* bytes

<column-constraint>

specifies to place an integrity constraint on the column.

Data source SAS data set, DB2 under UNIX and PC, MDS, MySQL, ODBC, Oracle, SAP HANA, Teradata

See [“<column-constraint> and <table-constraint> Arguments” on page 369](#)

[“Integrity Constraints” on page 374](#)

DEFAULT *value*

for each row that is added to the table, specifies a value for the column that is assumed when no other value is entered for that column.

value

specifies the default value.

Requirement *value* must be in the range of the specified data type.

Data source DB2 under UNIX and PC, MySQL, ODBC, Oracle, SAP HANA, Teradata

HAVING [FORMAT *format*][INFORMAT *informat*][LABEL '*label*']

specifies a clause that is used to associate a format, informat, or label with the column.

FORMAT *format*

specifies a SAS data format that is stored as column metadata. The format is not applied to the column data until execution time.

format

specifies a valid SAS data format. If the format is a format other than a FedSQL or DS2 format, it must be a valid format for the Base SAS.

See [Chapter 4, “FedSQL Formats,” on page 67,](#)

[Chapter 22, “DS2 Formats,” in *SAS DS2 Language Reference*](#)

Chapter 2, “Dictionary of Formats,” in *SAS Formats and Informats: Reference*

Restriction Formats that are created by PROC FORMAT cannot be stored as column metadata.

Note When you create a table, you can associate a SAS data format with a column. The format is not validated or applied to the column until execution time, such as in a PUT function. If the format is applied to the column in a Base SAS environment, any format can be stored with the column. If the format is applied to the column in the FedSQL environment, the format must be a valid FedSQL or DS2 format.

INFORMAT *informat*

specifies a SAS data informat that is stored as column metadata. The informat is not applied to the column data. No validation is done on the informat; it is only informational.

informat
specifies a SAS data informat.

See [“FedSQL Informats” on page 345](#)

Chapter 4, “Dictionary of Informats,” in *SAS Formats and Informats: Reference*

LABEL '*text-string*'

specifies a text string to use as an alternate column heading that appears in place of the column name in a query expression result.

Range 1 - 255 characters

Restriction A label can be created only with the CREATE TABLE statement and it cannot be used in a query expression operation.

Requirement *text-string* must be enclosed in either double or single quotation marks. If *text-string* contains a single quotation mark, enclose *text-string* in double quotation marks.

Data source SAS data set, SPD Engine data set

<column-constraint> and <table-constraint> Arguments

CONSTRAINT

begins a column constraint or a table constraint.

Data source SAS data set, DB2 under UNIX and PC, MDS, MySQL, ODBC, Oracle, SAP HANA, Teradata

constraint

specifies a name to identify the constraint.

See For constraint name requirements, see the documentation for your data source.

CHECK(*search-condition*)

specifies a condition for values in the column or table. *search-condition* is a valid FedSQL expression that resolves to a Boolean value. If *search-condition* is false, no changes are made to the table.

Restriction For a SAS data set, the search condition cannot contain FedSQL functions.

Data source SAS data set, Aster, DB2 under UNIX and PC, Greenplum, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

See [“FedSQL Expressions” on page 42](#)

PRIMARY KEY

specifies that, for each row in the table, the value in the column can be used to uniquely identify its respective row in the table. It is a value that does not change. A primary key cannot have a null value.

Data source SAS data set, Aster, DB2 under UNIX and PC, Greenplum, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

PRIMARY KEY (*column* [, ...*column*])

specifies that, for each row in the table, the values for all of the columns specified are used to uniquely identify its respective row in the table. A primary key cannot have a null value and it must be unique for each row in the column. It is a value that does not change.

Data source SAS data set, Aster, DB2 under UNIX and PC, Greenplum, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

UNIQUE

specifies that no two rows in the column can have the same value. A null value is allowed unless you specify NOT NULL.

Data source SAS data set, DB2 under UNIX and PC, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, Teradata

UNIQUE(*column* [, ...*column*])

specifies that no two rows in any of the specified columns can have the same value. A null value is allowed unless you specify NOT NULL.

Data source SAS data set, Aster, DB2 under UNIX and PC, Greenplum, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

NOT NULL

specifies that a null value is not valid in any row for the specified column.

Data source SAS data set, Aster, DB2 under UNIX and PC, Greenplum, MDS, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

FOREIGN KEY (*referencing-column* [, ...*referencing column*]) REFERENCES *referenced-table* (*referenced-column* [, ...*referenced-column* <*referential-trigger-action*>])

specifies the clause that relates columns in *table* to columns in another table through the values for those columns. Foreign keys must always include enough columns to uniquely identify a row in the referenced table. Foreign key constraints help ensure the integrity of related data in multiple tables.

FOREIGN KEY

begins the clause that identifies one or more columns in *table* that relate to columns in another table.

Data source Aster, DB2 under UNIX and PC, Greenplum, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

referencing-column

specifies the columns in *table* that relate to columns in another table.

REFERENCES

begins the clause that identifies the columns in another table.

referenced-table

specifies the table whose columns relate to columns in *table*.

referenced-column

specifies one or more columns in the referenced table that relate to the columns in *table*.

<referential-trigger-action>

specifies the action to take in the referenced table when columns in *table* are updated or deleted.

ON UPDATE <referential-action>

specifies the clause for the action to take when a column in *table* is updated.

ON DELETE <referential-action>

specifies the clause for the action to take when a column in *table* is deleted.

<referential-action>

specifies the action to take when a column in *table* is updated or deleted.

CASCADE

ON UPDATE, specifies to update all rows in *table* that reference an updated value in the referenced table. ON DELETE, removes all rows in *table* that reference the deleted rows in the referenced table.

SET NULL

specifies to set the referenced columns to a null value.

SET DEFAULT

specifies to set the referenced columns to their default value.

RESTRICT

specifies that if the referential constraint is not satisfied at any time while the FedSQL statement is processing, no action is taken and the update or deletion fails.

NO ACTION

specifies that if the referential constraint is not satisfied at the end of the FedSQL statement, no action is taken and the update or deletion fails.

<constraint-check-time>

specifies when the constraint is checked:

DEFERRABLE | NOT DEFERRABLE

specifies whether the constraint violation check can be performed after the transaction completes or if it must be checked at the end of each FedSQL statement. If DEFERRABLE is specified, the constraint check can occur at the end of the transaction. If NOT DEFERRABLE is specified, the constraint must be checked when the FedSQL statement terminates. Specifying DEFERRABLE is useful when more than one statement is necessary to complete a transaction.

Default DEFERRABLE**INITIALLY DEFERRED**

specifies that the constraint violation check is deferred, by default, at the beginning of each transaction and it does not occur until the end of a transaction.

INITIALLY IMMEDIATE

specifies that the constraint violation check occurs at the end of each FedSQL statement.

Details

Overview of the CREATE TABLE Statement

The CREATE TABLE statement enables you to create tables by defining table columns or by selecting columns from an existing table using a query expression. SAS provides extensions to the CREATE TABLE statement to support SAS data sources.

SAS Extensions for the CREATE TABLE Statement to Support SAS Data Sources

SAS extensions for the CREATE TABLE statement using FedSQL enable you to assign formats and informats to columns and add a label to a column.

SAS Formats

You can specify that the column data is stored and retrieved as a SAS formatted value by using the HAVING FORMAT clause. If the data type is not either CHAR or DOUBLE, it is converted to either CHAR or DOUBLE.

For example, to store the date in the format *ddmmmyy*, where *dd* is a two-digit year, *mmm* is a three-digit month, and *yy* is a two-digit year, the column definition would specify DOUBLE as the data type and FORMAT='DATE7.'. November 1, 2012 would be stored as 01Nov12. Your column definition might look like the following:

```
saleenddate double not null having format date7.;
```

In comparison, a column that stores date information using the DATE data type stores the date in the format *yyyy-mm-dd*. November 1, 2012 would be stored as 2012-11-01.

SAS Informats

SAS and user-defined informats can be stored and retrieved with the column data by using the HAVING INFORMAT clause. The informat is not applied to the data; it is information only. The client application is responsible for applying the informat to the data. Informats can be associated with all data types, but all data types will be converted to either CHAR or DOUBLE.

```
saledates double having informat anydtdte14.;
```

For more information about informats, see [“FedSQL Informats” on page 345](#).

Column Labels

A label is a descriptive, quoted, text string that is displayed in query expression results instead of the column name. You specify a label by using the LABEL argument in the HAVING clause:

```
saleenddate double not null having label 'Last Day of Sale';
```

A label cannot be used in a query expression operation.

Defining Columns for a Table

When you create columns for a table, the column name and the data type are required. All other column definition arguments are optional. For column naming conventions, see the documentation for your data source. FedSQL reserved words cannot be used as column names. For a list of FedSQL reserved words, see [“FedSQL Reserved Words” on page 61](#).

You can add an unspecified number of columns to a table by separating each column definition with a comma. Enclose the complete group of column definitions in parenthesis. The following CREATE TABLE statement creates the Customers table:

```
create table customers
( custid double primary key,
  name char(16),
  address char(64),
  city char(16),
  state char(2),
  country char(16),
  phone char(16),
  initorder date having label 'Initial Order'
);
```

Each column definition names the column and assigns a data type. The column ID contains the primary key integrity constraint, which is used to uniquely identify a table row. The INITORDER column is of the DATE data type, and when the column is displayed, the label **Initial Order** is used in place of the column name **initorder**.

You can add rows to the table by using the INSERT statement. The INSERT statement enables you to specify values for each of the table columns defined with the CREATE TABLE statement. For more information, see the [“INSERT Statement” on page 382](#).

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 column definition must equal the number of columns that are returned by the SELECT statement. If no column names are specified in the CREATE TABLE statement, the columns and default values that are returned by the SELECT statement are used in the new table.

This CREATE TABLE statement creates a new table that is based on only three columns from the CorpData table:

```
create table spainEmails
as select name, emailid, lastPurchaseDate from corpdata where country='Spain';
```

The following CREATE TABLE statement selects all columns from the CorpData table:

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

You can test the performance of a query expression before creating a table by using a table name of `_NULL_` for the query expression, as in this example:

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

FedSQL performs normal processing to create the table and the table appears to be created. The table is discarded when the query is complete.

Data Sources

When defining data types using the FedSQL language, in order for data to be stored, the defined data type must be available for data storage in that data source. Although FedSQL provides support for several data types, the data types that can be defined for a particular table depend on the data source. Each data source does not necessarily support all of the FedSQL data types.

In addition, data sources support variations of the standard SQL data types. That means that a data type that you specify might default to a different data type that might also have different attributes in the underlying data source. This is done when a data source does not natively support a specific data type, but data values of a similar data type can be converted without data loss. For example, to support the INTEGER data type, a SAS data set defaults the data type definition to SAS numeric, which is a DOUBLE. For details about data source implementation for each data type such as data source-dependent attributes, see [“Data Type Reference” on page 507](#).

Integrity Constraints

Integrity constraints are a set of data validation rules that you can specify to preserve the validity and consistency of your data. Integrity constraints that are specified in the CREATE TABLE statement are passed through to the data source. When a transaction modifies the table, the data source enforces integrity constraints.

A column constraint is a constraint that is defined for one column. A table constraint defines a constraint for two or more columns.

The following statements create SAS data sets using integrity constraints:

- Create a Products table using the product ID as the primary key; no two product IDs can identify the same product, and the product cannot be a null value. Check constraints at the end of a transaction:

```
create table products (prodid double primary key,
    product char(8) unique not null initially deferred);
```

- Create a Sales table with totals in US dollars, and include the country:

```
create table sales (prodid double,
    custid double primary key,
    totals double having format dollar10.,
    country char(30) not null);
```

- Create a SAS data set for customer credit card information. Customers are uniquely identified by their name and customer number.

```
create table custcredit (name char(30),
    custNum double,
    ccType char(15) having label 'Credit Card Type',
    ccNum char(20) having label 'Credit Card Number',
    ccExp date having label 'Expiration Date',
    CONSTRAINT ccconst primary key(name, CustNum));
```

When FedSQL encounters a DATE data type for a SAS data set, it does a type conversion from DATE to DOUBLE and assigns the DATE9. format to the column.

For information about constraints, see the documentation for your data source.

See Also

Statements

- [“ALTER TABLE Statement” on page 351](#)

- [“DROP TABLE Statement” on page 379](#)

CREATE VIEW Statement

Creates a view of data from one or more tables or other views.

Category: Data Definition

Data source: SAS data set, Aster, DB2 under UNIX and PC, Greenplum, HDMD, Hive, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Teradata

Syntax

CREATE VIEW *view* [**SECURITY** *security-type*] **AS** *query-expression*;

Arguments

SECURITY *security-type*

specifies the type of security that will be enforced for the view. *security-type* can be one of the following values.

DEFINER

specifies that the view is run with the schema owner’s credentials.

INVOKER

specifies that the view is run with the invoking user’s credentials.

Default If SECURITY is not used, security for the view defaults to INVOKER.

Restriction Security is available only with SAS Federation Server. The SECURITY *security-type* argument is ignored in a FedSQL request that is not directed to SAS Federation Server. For more information on the security available through SAS Federation Server, see the documentation for SAS Federation Server.

view

specifies the name of the view being created.

Requirement The view name must be different from any other view, table, or index in the same database.

query-expression

specifies the SELECT statement that retrieves the information from an existing table that is used to create the view.

See [“Creating and Populating Tables from a Query Expression” on page 373](#), [“Query Expressions and Subqueries” on page 42](#), and [“SELECT Statement” on page 386](#)

Details

Overview of the CREATE VIEW Statement

A *view* is a definition of a virtual table that is formed by a query expression against one or more tables or other views. A view contains no data. The view definition is stored as

metadata and consists of the view name, possibly the name of the columns in the view, and the query expression that is used to derive its rows.

If an underlying table or view in a view is deleted, SAS returns an error message when the view is executed. If the underlying table or view is changed, the view might have to be recreated.

Access to Views

To create a view, you must have CREATE VIEW privilege. As the view owner, you have all privileges for the view, including the ability to grant privileges to other users. To execute a view, the user must have SELECT privilege on the tables and views that are referenced in the view.

SELECT Statement Restrictions

The SELECT statement cannot include an ORDER BY clause.

See Also

Statements:

- [“CREATE TABLE Statement” on page 359](#)
- [“DESCRIBE VIEW Statement” on page 376](#)
- [“DROP VIEW Statement” on page 380](#)
- [“SELECT Statement” on page 386](#)

DESCRIBE VIEW Statement

Retrieves SQL from a view and returns a result set.

Category: Data Definition

Data source: SAS data set, Aster, DB2 under UNIX and PC, Greenplum, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Teradata

Syntax

DESCRIBE VIEW [*catalog*].[*schema*].*view*

DESCRIBE VIEW XML [*catalog*].[*schema*].*view*

Arguments

catalog

specifies the catalog that contains the view.

schema

specifies the schema that contains the view.

view

specifies the name of the view.

Details

Use DESCRIBE VIEW to retrieve standard SQL text from a view. Use DESCRIBE VIEW XML to retrieve SQL from a view as XML. This eliminates the need to implement a SQL parser on the client.

You must have ALTER VIEW privilege to run DESCRIBE VIEW on a secured view.

See Also

Statements:

- [“CREATE VIEW Statement” on page 375](#)

DELETE Statement

Deletes rows from a table.

Category: Data Manipulation

Supports: [“EXECUTE Statement” on page 381](#)

Data source: SAS data set, SPD Engine data set, Aster, DB2 under UNIX and PC, Greenplum, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

Syntax

DELETE FROM *table* [**WHERE** <search-condition>];

Arguments

table

specifies the table from which you are deleting rows.

Restriction If row-level permissions are in effect for the table, you cannot delete rows from the table. Row-level security is available only with SAS Federation Server. See the SAS Federation Server documentation for more information.

Tip You can use '.'*' after the table name for compatibility with Microsoft Access tables.

WHERE <search-condition>

specifies any valid WHERE clause used to limit the number of rows that are deleted.

Tip You can use parameter arrays in the WHERE statement.

See WHERE clause in the [“SELECT Statement” on page 386](#)

Details

The DELETE statement deletes the rows that satisfy the WHERE clause from the specified table. If you do not use a WHERE clause, all rows are deleted.

Comparisons

The DROP TABLE statement removes the table from the database completely, including the table structure. The DELETE statement deletes only the rows (data) in a table.

See Also

Statements:

- [“DROP TABLE Statement” on page 379](#)
- [“INSERT Statement” on page 382](#)
- [“UPDATE Statement” on page 412](#)

DROP INDEX Statement

Removes the specified index from a table.

Category: Data Definition

Supports: [“EXECUTE Statement” on page 381](#).

Data source: SAS data set, SPD Engine data set, Aster, DB2 under UNIX and PC, Greenplum, MDS, MySQL, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

Syntax

DROP INDEX *index* [FROM *table*] [FORCE];

Arguments

index

specifies the name of the index to be removed.

FROM *table*

specifies the name of the table where the index resides.

Requirements For a SAS data set, SPD Engine data set, and MDS table, you must include the FROM *table* syntax in order to specify the name of the table where the index resides.

For all data sources, you must include the FROM *table* syntax if you connected to the data source using a data source name (DSN) that has SAS security enabled.

FORCE

specifies that the index is removed without error processing. Use the FORCE keyword only when you are certain that removing the index without error processing will not negatively impact the table.

Details

If you drop a composite index, then the index is removed for all the columns that are named in that index.

Note: The DROP INDEX statement does not apply to indexes that were created by using the PRIMARY KEY or UNIQUE constraints of either the CREATE TABLE or ALTER TABLE statements, respectively. To remove those indexes, you must first remove the constraint by using the ALTER TABLE statement.

See Also

Statements:

- [“ALTER TABLE Statement” on page 351](#)
- [“CREATE INDEX Statement” on page 357](#)

DROP TABLE Statement

Removes a table from the database.

Category: Data Definition

Supports: [“EXECUTE Statement” on page 381](#)

Data source: SAS data set, SPD Engine data set, SASHDAT file, Aster, DB2 under UNIX and PC, Greenplum, HDMD, Hive, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

Syntax

DROP TABLE *table* [**FORCE**];

Arguments

table

specifies the name of the table to be removed.

Restriction You cannot alter or drop an MDS table while it is referenced in another transaction or statement. If a request fails, make sure other users are no longer using the table or have disconnected.

FORCE

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

Details

The DROP TABLE statement removes a table definition and all the table's data, metadata, and indexes.

If you use the DELETE statement to remove all the rows in a table, the table still exists until it is removed with the DROP TABLE statement.

You cannot use the DROP TABLE statement to remove a table that is referenced by a foreign key constraint. You must drop the foreign key constraint first, and then remove the table.

If you remove a table with indexed columns, then all the indexes are automatically removed. If you drop a composite index, then the index is removed for all the columns that are named in that index.

You should delete references in queries to any tables that you remove.

Teradata users: See [Appendix 6, “Usage Notes,” on page 557](#).

Comparisons

The DELETE statement only deletes the rows (data) in a table. The DROP TABLE statement removes the table from the database completely, including the table structure.

See Also

Statements:

- “CREATE TABLE Statement” on page 359
- “DELETE Statement” on page 377

DROP VIEW Statement

Removes a view from the database.

Category: Data Definition

Data source: SAS data set, Aster, DB2 under UNIX and PC, Greenplum, HDMD, Hive, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Teradata

Syntax

DROP VIEW *view* [**FORCE**];

Arguments

view

specifies the name of the view to be removed.

FORCE

specifies that the view is removed without error processing.

Details

A view can be considered a virtual table. The view is formed by running a query expression against one or more tables. Dropping a view does not change any data in the database. Only the metadata that is associated with the view is deleted.

Any view on a dropped table can be dropped explicitly by using the DROP VIEW statement.

Comparisons

The DROP TABLE statement removes a table from the database. The DROP VIEW statement removes a view from the database.

See Also

Statements:

- [“CREATE VIEW Statement” on page 375](#)
- [“DESCRIBE VIEW Statement” on page 376](#)
- [“DROP TABLE Statement” on page 379](#)

EXECUTE Statement

Sends a DBMS-specific SQL statement to a DBMS that FedSQL supports.

Category: FedSQL Pass-through

Data source: Aster, DB2 under UNIX and PC, Greenplum, HDMD, Hive, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

See: [“Data Source Connection” on page 12](#)
[“FedSQL Pass-Through Facility” on page 59](#)
 SQL documentation for your DBMS

Syntax

EXECUTE(*native-syntax*) **BY** *catalog*;

Arguments

native-syntax

specifies a native SQL query that can be run on the catalog’s driver. The EXECUTE statement accepts statements that produce a result set, as well as statements that do not produce a result set.

Restriction PROC FEDSQL does not support native syntax that contains embedded bare semicolons. You can, however, use the FedSQL language processor to run these statements.

Note The SQL statement might be case-sensitive, depending on your data source, and it is passed to the data source exactly as you enter it.

catalog

specifies the name of a catalog in the existing FedSQL connection.

Details

The EXECUTE statement is the only way that FedSQL provides to submit native SQL statements that do not produce a result set to a DBMS. Native SQL can be used in the following statements: DDLs, UPDATE, SELECT, and bulk loads.

A native connection for HDMD is a FedSQL connection.

Examples

Example 1: Using EXECUTE to Create and Drop a Table

```
execute(create table t_blob (col1 blob)) by saphana;
execute(drop table t_blob) by saphana;
```

Example 2: Using EXECUTE to Produce a Result Set

```
execute(select i, rank() over (order by j) rank from table_a ) by oracle;
```

See Also

Concepts:

- [“FedSQL Pass-Through Facility” on page 59](#)

Statements:

- [“ALTER TABLE Statement” on page 351](#)
- [“BEGIN Statement” on page 355](#)
- [“COMMIT Statement” on page 356](#)
- [“CREATE INDEX Statement” on page 357](#)
- [“CREATE TABLE Statement” on page 359](#)
- [“DELETE Statement” on page 377](#)
- [“DROP INDEX Statement” on page 378](#)
- [“DROP TABLE Statement” on page 379](#)
- [“INSERT Statement” on page 382](#)
- [“ROLLBACK Statement” on page 385](#)
- [“SELECT Statement” on page 386](#)
- [“UPDATE Statement” on page 412](#)

INSERT Statement

Adds rows to a specified table.

Category: Data Manipulation

Supports: [“EXECUTE Statement” on page 381](#)

Data source: SAS data set, SPD Engine data set, SASHDAT file, Aster, DB2 under UNIX and PC, Greenplum, HDMD, Hive, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

Syntax

Form 1: **INSERT INTO** *table*
 {VALUES (*value* | NULL [, ...*value* | NULL])};

- Form 2: **INSERT INTO** *table*
 { (*column* [, ...*column*]) VALUES (*value* | NULL [, ...*value* | NULL]) };
- Form 3: **INSERT INTO** *table*
 { (*column* [, ...*column*]) [*query-expression*] };

Arguments

table

specifies the name of a table into which you are inserting rows.

Restriction If row-level permissions are in effect for the table, you cannot insert rows into the table. Row-level security is available only with SAS Federation Server. For more information about row-level security, see the SAS Federation Server documentation.

value

specifies a data value to insert into the table.

Restriction There must be one *value* for each column in the table or for each column in the column list (*column*).

Requirement If columns are specified in the column list (*column*), the data values list should be in the same order as the column list.

Tip You can use parameter arrays in the INSERT statement.

column

specifies the column into which you are inserting data.

Tip You can specify more than one column. The columns do not have to be in the same order as they appear in the table as long as the order of the column list and the data values list match one another.

<query-expression>

specifies any valid query expression that returns rows and where the number of columns in each row is the same as the number of items to be inserted.

See [“SELECT Statement” on page 386](#)

Details

Form 1: INSERT Statement without Column Names

This form of the INSERT statement that uses the VALUES clause without specifying column names can be used to insert lists of values into a table. You specify a value for each column in the table. One row is inserted for each VALUES clause. The order of the values in the VALUES clause should match the order of the columns in the table.

The following code fragment shows how the INSERT statement could be used with the VALUES clause to insert data values into the Customers table that was created in [“CREATE TABLE Statement” on page 359](#). The Customers table has columns CustId, Name, Address, City, State, Country, PhoneNumber, and InitOrder.

```
insert into customers values (1,'Peter Frank', '300 Rock Lane', 'Boulder', 'CO',
'United States', '3039564321', date '2012-01-14');
insert into customers values (2,'Jim Stewart', '1500 Lapis Lane', 'Little Rock',
'AR', 'United States', '8705553978', date '2012-03-20');
```

```

insert into customers values (3,'Janet Chien', '75 Jujitsu', 'Nagasaki', '', 'Japan',
'01181956879932', date '2012-06-07');
insert into customers values (4,'Qing Ziao', '10111 Karaje', 'Tokyo', '', 'Japan',
'0118136774351', date '2012-10-12');
insert into customers values (5,'Humberto Sertu', '876 Avenida Blanca', 'Buenos Aires',
'', 'Argentina','01154118435029', date '2012-12-15');

```

These statements add data values to the Sales table that was created in the “[CREATE TABLE Statement](#)” on page 359. The Sales table has columns ProdID, CustID, Totals, and Country.

```

insert into sales values (3234, 1, 189400, 'United States');
insert into sales values (1424, 3, 555789, 'Japan');
insert into sales values (3421, 4, 781183, 'Japan');
insert into sales values (3421, 2, 2789654, 'United States');
insert into sales values (3975, 5, 899453, 'Argentina');

```

Note that character values are enclosed in quotation marks.

Form 2: INSERT Statement with Column Names

This form of the INSERT statement that uses the VALUES clause with specific column names can also be used to insert lists of values into a table. You give values for the columns specified in the list of column names. The column names do not have to be in the same order as they appear in the table, but the order of the values in the VALUES clause must match the order of the column names in the INSERT column list. One row is inserted for each VALUES clause.

In the following code fragment, column names are used to add values to the Products table that was created in the “[CREATE TABLE Statement](#)” on page 359..

```

insert into products (prodid, product) values (1424, 'Rice');
insert into products (prodid, product) values (3421, 'Corn');
insert into products (prodid, product) values (3234, 'Wheat');
insert into products (prodid, product) values (3485, 'Oat');

```

The column list enables you to add a row that contains values for specified columns. If you do not specify the column list, values for every column in the table must be provided by the VALUES clause.

Form 3: INSERT Statement with a Query Expression

This form of the INSERT statement uses a query expression that inserts rows that were selected from another table or tables. The order of the values returned by the query expression should match the order of the columns that are specified in the list of column names. All the rows that are returned by the query expression are inserted into the table. The following restrictions should be noted when you use this form of the INSERT statement.

- If you do not specify the column list, values for every column in the table must be provided by the query expression.
- If you do not specify all columns in the column list, a null value is inserted for each column that is not listed.
- If the data type of the column that is being added does not match the data type of the column in the table, an automatic type conversion is attempted.

Here is an example of an INSERT statement that contains a query expression.

```

insert into salesum (

```

```

        sum_sales,
        max_sale,
        min_sale,
        avg_sale)
values (
    (select sum(totals) from sales),
    (select max(totals) from sales),
    (select min(totals) from sales),
    (select avg(totals) from sales)
);

```

This example uses subqueries to populate a new table, SalesSum, with aggregate values from the Sales table. A subquery returns only one row. To see the content of the table, see [“Sales” on page 541](#). For more information, see [“Overview of Subqueries” on page 44](#).

Inserting Date, Time, and Timestamp Values

For information about inserting date, time, and timestamp values, see [“Dates and Times in FedSQL” on page 51](#).

Comparisons

The DELETE statement enables you to delete rows from a table. The UPDATE statement enables you to change rows in a table. The INSERT statement enables you to insert new rows into a table.

See Also

Statements:

- [“DELETE Statement” on page 377](#)
- [“SELECT Statement” on page 386](#)
- [“UPDATE Statement” on page 412](#)

ROLLBACK Statement

Rolls back transaction changes to the beginning of the transaction.

Category:	Data Control
Restriction:	The ROLLBACK statement has an effect only when autocommit functionality is off.
Supports:	“EXECUTE Statement” on page 381
Data source:	Greenplum, MDS, MySQL, ODBC, Oracle, SAP HANA, Sybase IQ, Teradata

Syntax

```
ROLLBACK [TRANSACTION];
```

Details

When your program has completed all of the statements in the transaction, you must explicitly terminate the transaction using COMMIT or ROLLBACK. You use a

ROLLBACK statement to roll back, or undo, the changes that have been made since the start of the transaction.

You cannot roll back the changes to the database after a COMMIT statement is executed.

The ROLLBACK statement has an effect only when autocommit functionality is off. For most data sources, autocommit functionality is on by default. See the server administration documentation for information about how to turn off autocommit functionality. For example, see *SAS Federation Server: Administrator's Guide* for the appropriate connection option to the FedSQL driver. For the FEDSQL procedure, see *Base SAS Procedures Guide*.

Note: ROLLBACK has no effect on the SASHDAT data source.

Comparisons

The COMMIT statement takes all the changes that have been performed since the start of the transaction and makes them a permanent part of the database. The ROLLBACK statement causes all the changes that were made by the transaction to be rolled back to the start of the transaction.

See Also

Statements:

- [“BEGIN Statement” on page 355](#)
- [“COMMIT Statement” on page 356](#)

SELECT Statement

Retrieves columns and rows of data from tables.

Categories: Data Definition
Data Manipulation

Supports: [“EXECUTE Statement” on page 381](#)

Data source: SAS data set, SPD Engine data set, Aster, DB2 under UNIX and PC, Greenplum, HDMD, Hive, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP, SAP HANA, Sybase IQ, Teradata

Syntax

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

```

SELECT <select-list>
    FROM <table-specification>
    [WHERE <search-condition>]
    [GROUP BY <grouping-column>]
    [HAVING <search-condition>]
    [ORDER BY <sort-specification>]
    [LIMIT {count | ALL}]
    [OFFSET number]
;

```

The detailed syntax of the SELECT statement is as follows.

```

<query-expression>
    [ORDER BY <sort-specification> [, ...<sort-specification>]];

<query-expression>::=
    {<query-specification> | <query-expression>}
    {UNION [ALL] | EXCEPT | INTERSECT} {<query-specification> | <query-expression>}

<query-specification>::=
    SELECT [ALL | DISTINCT] <select-list> <table-expression>

<select-list>::=
    *
    | column [AS column-alias]
    | expression [AS column-alias]
    | table.*
    | table-alias.*

<table-expression>::=
    FROM <table-specification> [, ...<table-specification>]
    [WHERE <search-condition>]
    [GROUP BY <grouping-column> [, ...<grouping-column>]]
    [HAVING <search-condition>]

<table-specification>::=
    table [[AS] alias]
    | CONNECTION TO catalog (<native-syntax>) [[AS] table-alias]
    | (<query-specification>) [AS] alias
    | <joined-table>

```

```

<joined-table>::=
    <cross-join>
    | <qualified-join>
    | <natural-join>
    <cross-join>::=
        <table-specification> CROSS JOIN <table-specification>
    <qualified-join>::=
        <table-specification> [<join-type>] JOIN <table-specification> <join-specification>
    <natural-join>::=
        <table-specification> NATURAL [<join-type>] JOIN <table-specification>
    <join-type>::=
        INNER
        | LEFT [OUTER]
        | RIGHT [OUTER]
        | FULL [OUTER]
    <join-specification>::=
        ON <search-condition>
        | USING (column [, ...column])
    <search-condition>::=
        {
            [NOT] {<sql-expression> | (<search-condition>)}
            [{AND | OR} [NOT] {<sql-expression> | (<search-condition>)}]
        }
        [,... {[NOT] {<sql-expression> | (<search-condition>)}
            [{AND | OR} [NOT] {<sql-expression> | (<search-condition>)}}]
    <sql-expression>::=
        expression {operator | predicate} expression
    <sort-specification>::=
        {order-by-expression [ASC | DESC]} [, ...order-by-expression [ASC | DESC]]
    <grouping-column>::=
        column [, ...column]
        | column-position-number
        | <sql-expression>

```

Arguments

See the following sections for syntax argument descriptions.

- “SELECT Clause” on page 389
- “FROM Clause” on page 391
- “WHERE Clause” on page 400
- “GROUP BY Clause” on page 400
- “HAVING Clause” on page 401
- “ORDER BY Clause” on page 402
- “UNION Operator” on page 407

- “LIMIT Clause” on page 406
- “OFFSET Clause” on page 407
- “EXCEPT Operator” on page 408
- “INTERSECT Operator” on page 409

Details

Overview

The SELECT statement can be used in two ways.

- The single row SELECT statement, which can be executed by itself, returns only one row. For example:

```
select 42;
select 42 as x;
```

The first code fragment returns a single column that contains the value 42. The column is named “column”. The second code fragment returns a similar column. However, the column is named “x”.

- A query specification begins with the SELECT keyword (called a SELECT clause) and cannot be used by itself. It reads column values from one or more tables and enables you to define conditions for the data that will be returned from the tables. It must be used as a part of another SQL statement and can return more than one row. A query specification creates a virtual table. For example:

```
select column(s)
from table(s)
where condition(s);
```

The order of clauses in the SELECT statement is important. The optional clauses can be omitted but, when used, they must appear in the appropriate order. A SELECT statement can be specified within a SELECT statement (called a subquery). The ORDER BY, OFFSET, and LIMIT clauses can be used only on the outermost SELECT of a SELECT statement.

A view can be specified in the SELECT statement wherever a table can be specified. FedSQL supports native DBMS views and FedSQL views. It does not support PROC SQL views.

Note: There is no limit on the number of tables that you can reference in a FedSQL query. However, queries with a large number of table references can cause performance issues.

SELECT Clause

Description

Lists the columns that will appear in a virtual result table.

Syntax

SELECT [[ALL](#) | [DISTINCT](#)] <select-list>

```

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

```

Arguments

ALL

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

DISTINCT

eliminates duplicate rows in the result table.

<select-list>

specifies the columns to be selected for the result table.

*

selects all columns in the table that is listed in the FROM clause.

column-alias

assigns a temporary, alternate name to the column.

column [AS *column-alias*]

selects a single column. When [AS *column-alias*] is specified, assigns the column alias to the column.

<query-specification>

specifies an embedded SELECT subquery.

See [“Overview of Subqueries” on page 44](#)

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

derives a column name from an expression.

See [“<sql-expression>” on page 339](#)

*table.**

selects all columns in the table.

*table-alias.**

selects all columns in the table.

See [“Table Aliases” on page 393](#)

Asterisk (*) Notation

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

Column Aliases

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

The keyword AS is required to distinguish a column alias from other column names.

Column aliases are optional, and each column name in the SELECT clause can have an alias. After you assign an alias to a column, you can use the alias to refer to that column in other clauses.

FROM Clause

Description

Optionally specifies source tables.

Syntax

FROM <table-specification> [, ...<table-specification>]

<table-specification>::=

table [[AS] *table-alias*]

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

| (<query-specification>) [AS] *alias*

| <joined-table>

<joined-table>::=

<cross-join>

| <qualified-join>

| <natural-join>

<cross-join>::=

<table-specification> CROSS JOIN <table-specification>

<qualified-join>::=

<table-specification> [<join-type>] JOIN <table-specification> <join-specification>

<natural-join>::=

<table-specification> NATURAL [<join-type>] JOIN <table-specification>

<join-type>::=

INNER

| LEFT [OUTER]

| RIGHT [OUTER]

| FULL [OUTER]

<join-specification>::=

ON <search-condition>

| USING (*column* [, ...*column*])

Arguments

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

specifies data from a DBMS catalog by using the SQL pass-through facility. You can use SQL syntax that the DBMS understands, even if that syntax is not valid in FedSQL. For more information, see [“FedSQL Pass-Through Facility” on page 59](#).

CROSS JOIN

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

See [“Cross Joins” on page 395](#)

JOIN

defines a join that enables you to filter the data by using a search condition or by using specific columns.

See [“Qualified Joins” on page 395](#)

NATURAL JOIN

defines a join that selects rows from two tables that have equal values in columns that share the same name and the same type.

See [“Natural Joins” on page 399](#)

(<query-specification>) [AS] *alias*

specifies an embedded SELECT subquery that functions as an in-line view. *alias* defines a temporary name for the in-line view and is required. An in-line view saves you a programming step. Rather than creating a view and referring to it in another query, you can specify the view in-line in the FROM clause.

See [“Overview of Subqueries” on page 44](#)

table

specifies the name of a table.

table-alias

specifies a temporary, alternate name for *table*. The AS keyword is optional.

INNER

specifies that only the subset of rows from the first table that matches rows from the second table are returned. Unmatched rows from both tables are discarded.

LEFT [OUTER]

specifies that matching rows and rows from the first table that do not match any row in the second table are returned.

RIGHT [OUTER]

specifies that matching rows and rows from the second table that do not match any row in the first table are returned.

FULL [OUTER]

specifies that all matching and unmatching rows from the first and second table are returned.

column

specifies the name of a column.

ON <search-condition>

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

See [“<search-condition>” on page 410](#)

USING (*column* [...*column*])

specifies which columns to use in an inner or outer join.

See [“ON and USING Clauses” on page 398](#)

Overview

The FROM clause enables you to specify source tables. You can reference tables by specifying their table name, by submitting a query to a DBMS, by specifying an embedded SELECT subquery, or by specifying a join.

Table Aliases

A table alias is a temporary, alternate name for a table. Table aliases are used in joins to distinguish the columns of one table from those in the other table or tables and can make a query easier to read by abbreviating the table names. A table name or alias must be prefixed to a column name when you are joining tables that have matching column names. Column names in *reflexive joins* (joining a table with itself) must be prefixed with a table alias in order to distinguish which copy of the table the column comes from. A table alias cannot be given an alias.

Joined Tables

When multiple table specifications are listed in the FROM clause, they are processed to form one table. The result table contains data from each contributing table. These queries are referred to as *joins*. Joins do not alter the original table.

Conceptually, when two tables are specified, each row of table A is matched with all the rows of table B to produce an internal or intermediate table. The number of rows in the intermediate table (*Cartesian*) is equal to the product of the number of rows in each of the source tables. The intermediate table becomes the input to the rest of the query in which some of its rows can be eliminated by the WHERE, ON, or USING clause or summarized by a function.

For an overview of FedSQL join operations, see [“Join Operations” on page 27](#).

Specifying the Rows to Be Returned

The WHERE, ON, and USING clauses contain the conditions under which the rows in the Cartesian product are kept or eliminated in the result table. WHERE is used to select rows from inner joins. ON is used to select rows from inner or outer joins. USING is used to select specific columns to be included in the join. The condition is evaluated for each row from each table in the intermediate table described in [“Joined Tables” on page 393](#). The row is considered to be a match if the result of the expression is true (a nonzero, nonmissing, or null value) for that row.

Simple Joins

The most basic type of join is simply a list of multiple tables, separated by commas, in the FROM clause of a SELECT statement. The following query joins the two tables GrainProducts and Sales that are shown in [“Tables Used in Examples” on page 537](#).

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

Output 8.1 Simple Join - GrainProducts and Sales Table

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

Joining tables in this way returns the *Cartesian* of the tables. Each row from the first table is combined with every row of the second table. The number of rows in the result table is equal to the number of rows in the first table multiplied by the number of rows in the second table.

The Cartesian product of a simple join can result in large, meaningless tables. You can subset a simple join by using a WHERE clause. This type of simple join is known as an *equijoin*. The following query subsets the previous table by matching the ID columns and creates the table shown in [Output 8.2 on page 395](#).

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

In an equijoin, the comparison has to be an equality comparison. Multiple match criteria (not shown here) can be specified by using the AND operator. When multiple match criteria are specified, only rows meeting all of the equality tests are returned.

Output 8.2 *Equijoin - GrainProducts and Sales Table*

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

Cross Joins

The cross join functions the same as a simple join; it returns the product of two tables. Like a Cartesian product, a cross join's output can be limited by a WHERE clause.

The following queries produce the same result.

```
select * from grainproducts, sales;
select * from grainproducts cross join sales;
```

Note: Do not use an ON clause with a cross join. An ON clause causes a cross join to fail. However, you can use a WHERE clause to subset the output.

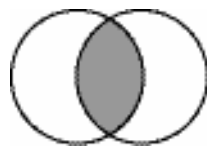
Qualified Joins

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

The two types of qualified joins are inner and outer.

Inner Joins

Figure 8.1 *Inner Join Diagram*



An *inner join* returns a result table for all the rows in one table that have one or more matching rows in another table. Using the GrainProducts and Sales tables, the following query matches the product ID columns of the two tables and creates the result table shown in [Output 8.3 on page 396](#).

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

Output 8.3 Inner Join - GrainProducts and Sales Table

Inner Join - GrainProducts and Sales					
PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
3234	Wheat	3234	1	\$189,400	United States
1424	Rice	1424	3	\$555,789	Japan
3421	Corn	3421	4	\$781,183	Japan
3421	Corn	3421	2	\$2,789,654	United States

You can use the ON or USING clause instead of the WHERE clause to specify the column or columns on which you are joining the tables. However, you can continue to use the WHERE clause to subset the query result.

Note that an inner join with an ON or USING clause can provide the same functionality as listing tables in the FROM clause and specifying join columns with a WHERE clause (an equijoin). For example, these two sets of code use the inner join construction.

```
select *
  from grainproducts inner join sales
    on grainproducts.prodId=sales.prodId;

select *
  from grainproducts inner join sales
    using (prodId);
```

This code produces the same output as the previous code but uses the inner join construction.

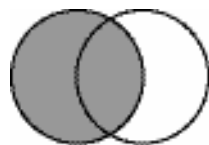
```
select *
  from grainproducts, sales
    where grainproducts.prodId=sales.prodId;
```

Outer Joins

Outer joins are inner joins that have been augmented with rows from one table that do not match with any row from the other table in the join. The result table includes rows that match and rows that do not match from the join's source tables. Nonmatching rows have null or missing values in the columns from the unmatched table. You can use the ON or USING clause instead of the WHERE clause to specify the column or columns on which you are joining the tables. However, you can continue to use the WHERE clause to subset the query result.

The three types of outer joins are left, right, and full.

Left Outer Joins

Figure 8.2 Left Outer Join Diagram

A left outer join lists matching rows and rows from the first table listed in the FROM clause that do not match any row in the second table listed in the FROM clause. Using the GrainProducts and Sales tables, the following code creates a table with matching

rows from the GrainProducts and Sales tables and the unmatched rows from the GrainProducts table. Note that missing values are shown for Sales table data in the unmatched row from the GrainProducts table.

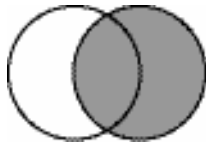
```
title 'Left Outer Join - GrainProducts and Sales';
select *
  from grainproducts left outer join sales
    on grainproducts.prodid=sales.prodid;
```

Output 8.4 Left Outer Join - GrainProducts and Sales Table

Left Outer Join - GrainProducts and Sales					
PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
3234	Wheat	3234	1	\$189,400	United States
1424	Rice	1424	3	\$555,789	Japan
3421	Corn	3421	4	\$781,183	Japan
3421	Corn	3421	2	\$2,789,654	United States
3485	Oat	-	-	-	-

Right Outer Joins

Figure 8.3 Right Outer Join Diagram

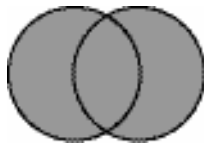


A right outer join lists matching rows and rows from the second table listed in the FROM clause that do not match any row in the first table listed in the FROM clause. Using the GrainProducts and Sales tables, the following code creates a table with matching rows from the GrainProducts and Sales tables and the unmatched rows from the Sales table. Note that missing values are shown for GrainProducts table data in the unmatched row from the Sales table.

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

Output 8.5 Right Outer Join - GrainProducts and Sales Table

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

Full Outer Joins**Figure 8.4** Full Outer Join Diagram

A full outer join combines the left outer join and the right outer join. The result table contains both the matching and unmatched rows from the left and right tables. Using the GrainProducts and Sales tables, the following code creates a table with matching rows from the GrainProducts and Sales tables and the unmatched rows from the GrainProducts and Sales tables. Note that missing values are shown for data in the unmatched rows.

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

Output 8.6 Full Outer Join - GrainProducts and Sales Table

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

ON and USING Clauses

You can use an ON clause with an expression that specifies a condition on which the join is based. The conditional expression can contain any predicate, although column

names and comparison operators are most often used. The ON clause with an inner join is equivalent to a WHERE clause. The ON clause with an outer join (left, right, or full) is different from a WHERE clause. The ON clause with an outer join filters the rows and then includes the nonmatched rows with the null or missing values.

You can use a USING clause to specify one of two columns to include in the result table.

The difference between the ON clause and the USING clause is that you can use any conditional expression with the ON clause. The USING clause always implies an equality between the column names. For example, this ON clause eliminates **United States** from the results table.

```
title 'Inner Join - GrainProducts and Sales Outside US';
select *
  from grainproducts inner join sales
    on sales.country <> 'United States'
     AND grainproducts.prodid=sales.prodid;
```

Output 8.7 Inner Join - GrainProducts and Sales outside the US

Inner Join - GrainProducts and Sales Outside US					
PRODID	PRODUCT	PRODID	CUSTID	TOTALS	COUNTRY
1424	Rice	1424	3	\$555,789	Japan
3421	Corn	3421	4	\$781,183	Japan

Natural Joins

A *natural join* selects rows from two tables that have equal values in columns that share the same name and the same type. An error results if two columns have the same name but different types. You can perform an inner, left, right, or full natural join. If *join-type* is omitted when specifying a natural join, then INNER is implied. If like columns are not found, then a cross join is performed. You can use a WHERE clause to limit the output.

Using the GrainProducts and Sales tables, the following code performs a natural left outer join.

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

Output 8.8 Natural Left Outer Join - GrainProducts and Sales Table

Natural Left Outer Join - GrainProducts and Sales				
PRODID	PRODUCT	CUSTID	TOTALS	COUNTRY
3234	Wheat	1	\$189,400	United States
1424	Rice	3	\$555,789	Japan
3421	Corn	4	\$781,183	Japan
3421	Corn	2	\$2,789,654	United States
3485	Oat	.	.	.

Notice that the *prodid* column appears only once in the result table.

Note: Do not use an ON clause with a natural join. An ON clause causes a natural join to fail. When using a natural join, an ON clause is implied, matching all like columns.

WHERE Clause

Description

Subsets the result table based on the specified search conditions.

Syntax

WHERE <search-condition>

Arguments

<search-condition>

specifies the conditions for the rows returned by the WHERE clause.

See “<search-condition>” on page 410

Details

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

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

```
where max(inventory1)>10000;
```

However, you can use this WHERE clause.

```
where max(inventory1, inventory2)>10000;
```

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

GROUP BY Clause

Description

Specifies how to group the data for summarizing.

Syntax

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

<grouping-column>::=

column [, ...*column*]

| *column-position-number*

| <sql-expression>

Arguments

column

specifies the name of a column or a column alias.

column-position-number

specifies a nonnegative integer that equates to a column position.

<sql-expression>

specifies a valid SQL expression.

See [“<sql-expression>” on page 339](#)

Details

The GROUP BY clause groups data by a specified column or columns.

If the column or columns on which you are grouping contain missing or null values in some rows, SAS collects all the rows with missing or null values in the grouping columns into a single group.

You can specify more than one grouping column to get more detailed reports. If more than one grouping column is specified, then the first one determines the major grouping.

Integers can be substituted for column names in the GROUP BY clause. For example, if the grouping column is 2, then the results are grouped by values in the second column. Note that if you use a floating-point value (for example, 2.3) instead of an integer, then FedSQL ignores the decimal portion.

You can group the output by the values that are returned by an expression. For example, if X is a numeric variable, then the output of the following is grouped by the values of X.

```
select x, sum(y)
  from table1
 group by x;
```

Similarly, if Y is a character variable, then the output of the following is grouped by the values of Y.

```
select sum(x), y
  from table1
 group by y;
```

When you use a GROUP BY clause, you can also use an aggregate function in the SELECT clause or in a HAVING clause to instruct SAS in how to summarize the data for each group. When you use a GROUP BY clause without an aggregate function, SAS treats the GROUP BY clause as if it were an ORDER BY clause.

You can use the ORDER BY clause to specify the order in which rows are displayed in the result table. If you do not specify the ORDER BY clause, groups returned by the GROUP BY clause are not in any particular order.

Note: For an SPD Engine data set, utility files are used for certain operations that need extra space. The GROUP BY clause requires a utility file and the Base SAS UTILLOC system option allocates space for that utility file. For more information, see *SAS Scalable Performance Data Engine: Reference*.

Note: FedSQL does not support remerging of summary statistics.

HAVING Clause**Description**

Subsets grouped data based on specified search conditions.

Syntax

HAVING <search-condition>

Arguments

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

See “<search-condition>” on page 410

Details

The HAVING clause requires a search condition (one or more expressions separated by an operand or predicate) that specifies which rows are chosen for inclusion in the result table. A HAVING clause evaluates as either true or false for each group in a query. You can use a HAVING clause with a GROUP BY clause to filter grouped data. The HAVING clause affects groups in a way that is similar to how a WHERE clause affects individual rows.

Queries that contain a HAVING clause usually also contain a GROUP BY clause, an aggregate function, or both. When you use a HAVING clause without a GROUP BY clause, SAS treats the HAVING clause as if it were a WHERE clause.

Table 8.1 Differences between the HAVING Clause and WHERE Clause

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

ORDER BY Clause

Description

Specifies the order in which rows are returned in a result table.

Syntax

ORDER BY <sort-specification> [, ...<sort-specification>];

<sort-specification>::=

 {*order-by-expression* [ASC | DESC]} [, ...*order-by-expression* [ASC | DESC]]

 | {*order-by-expression* [COLLATE *collating-sequence-options*]} [, ...*order-by-expression* [COLLATE *collating-sequence-options*]]

Arguments

order-by-expression
specifies a column on which to sort. The sort column can be one of the following.

column

specifies the name of a column or a column alias.

column-position-number

specifies a nonnegative integer that equates to a column position.

<sql-expression>

specifies any valid SQL expression.

See [“<sql-expression>” on page 339](#)

ASC

orders the data in ascending order. This is the default order; if ASC or DESC are not specified, the data is ordered in ascending order.

DESC

orders the data in descending order.

COLLATE *collating-sequence-options*

specifies linguistic collation, which sorts characters according to rules of the specified language. The rules and default collating sequence options are based on the language specified in the current locale setting. The implementation is provided by the International Components for Unicode (ICU) library and produces results that are largely compatible with the Unicode Collation Algorithms (UCA).

The *collating-sequence-options* argument can be one of the following values:

DANISH | FINNISH | ITALIAN | NORWEGIAN | POLISH | SPANISH |
SWEDISH

sorts characters according to the language specified.

LINGUISTIC [*collating-rules*]

collating-rules can be one of the following values:

ALTERNATE_HANDLING=SHIFTED

controls the handling of variable characters like spaces, punctuation, and symbols. When this option is not specified (using the default value NON_IGNOREABLE), differences among these variable characters are of the same importance as differences among letters. If the ALTERNATE_HANDLING option is specified, these variable characters are of minor importance.

Default NON_IGNOREABLE

Tip

The SHIFTED value is often used in combination with STRENGTH= set to Quaternary. In such a case, spaces, punctuation, and symbols are considered when comparing strings, but only if all other aspects of the strings (base letters, accents, and case) are identical.

CASE_FIRST=

specify order of uppercase and lowercase letters. This argument is valid for only TERTIARY, QUATERNARY, or IDENTICAL levels. The following table provides the values and information for the CASE_FIRST argument:

Table 8.2 CASE_FIRST= Values

Value	Description
UPPER	Sorts uppercase letters first, then the lowercase letters.

LOWER	Sorts lowercase letters first, then the uppercase letters.
-------	--

COLLATION=

The following table lists the available COLLATION= values. If you do not select a collation value, then the user's locale-default collation is selected.

Table 8.3 COLLATION= Values

Value	Description
BIG5HAN	specifies Pinyin ordering for Latin and specifies big5 charset ordering for Chinese, Japanese, and Korean characters.
DIRECT	specifies a Hindi variant.
GB2312HAN	specifies Pinyin ordering for Latin and specifies gb2312han charset ordering for Chinese, Japanese, and Korean characters.
PHONEBOOK	specifies a telephone-book style for ordering of characters. Select PHONEBOOK only with the German language.
PINYIN	specifies an ordering for Chinese, Japanese, and Korean characters based on character-by-character transliteration into Pinyin. This ordering is typically used with simplified Chinese.
POSIX	is the Portable Operating System Interface. This option specifies a "C" locale ordering of characters.
STROKE	specifies a nonalphabetic writing style ordering of characters. Select STROKE with Chinese, Japanese, Korean, or Vietnamese languages. This ordering is typically used with Traditional Chinese.
TRADITIONAL	specifies a traditional style for ordering of characters. For example, select TRADITIONAL with the Spanish language.

LOCALE= *locale_name*

specifies the locale name in the form of a POSIX name (for example, ja_JP). For more information, see *SAS National Language Support (NLS): Reference Guide*.

NUMERIC_COLLATION=

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

Table 8.4 NUMERIC_COLLATION= Values

Value	Description
ON	Order numbers by the numeric value. For example, "8 Main St." would sort before "45 Main St."

OFF Order numbers by the character value. For example, "45 Main St." would sort before "8 Main St.".

Default OFF

STRENGTH=

The value of strength is related to the collation level. There are five collation-level values. The following table provides information about the five levels. The default value for strength is related to the locale.

Table 8.5 STRENGTH= Values

Value	Type of Collation	Description
PRIMARY or 1	PRIMARY specifies differences between base characters (for example, "a" < "b").	It is the strongest difference. For example, dictionaries are divided into different sections by base character.
SECONDARY or 2	Accents in the characters are considered secondary differences (for example, "as" < "às" < "at").	A secondary difference is ignored when there is a primary difference anywhere in the strings. Other differences between letters can also be considered secondary differences, depending on the language.
TERTIARY or 3	Upper and lowercase differences in characters are distinguished at the tertiary level (for example, "ao" < "Ao" < "aò").	A tertiary difference is ignored when there is a primary or secondary difference anywhere in the strings. Another example is the difference between large and small Kana.
QUATERNARY or 4	When punctuation is ignored at level 1–3, an additional level can be used to distinguish words with and without punctuation (for example, "ab" < "a-b" < "aB").	The quaternary level should be used if ignoring punctuation is required or when processing Japanese text. This difference is ignored when there is a primary, secondary, or tertiary difference.
IDENTICAL or 5	When all other levels are equal, the identical level is used as a tiebreaker. The Unicode code point values of the Normalization Form D (NFD) form of each string are compared at this level, just in case there is no difference at levels 1–4.	This level should be used sparingly, as only code point values differences between two strings is an extremely rare occurrence. For example, only Hebrew cantillation marks are distinguished at this level.

Alias LEVEL=

Restriction	Linguistic collation is not supported on platforms VMS on Itanium (VMI) or 64-bit Windows on Itanium (W64).
Tip	The <i>collating-rules</i> must be enclosed in parentheses. More than one collating rule can be specified.
See	“ICU License - ICU 1.8.1 and later” on page 559. <hr/> The section on Linguistic Collation in <i>SAS National Language Support (NLS): Reference Guide</i> . <hr/> Refer to http://www.unicode.org for the Unicode Collation Algorithm (UCA) specification.

Details

The ORDER BY clause sorts the result of a query expression according to the order specified in that query. When this clause is used, the default ordering sequence is ascending, from the lowest value to the highest.

If an ORDER BY clause is omitted, then a particular order to the output rows, such as the order in which the rows are encountered in the queried table, cannot be guaranteed. Without an ORDER BY clause, the order of the output rows is determined by the internal processing of FedSQL, the default collating sequence of SAS, and your operating environment. Therefore, if you want your result table to appear in a particular order, then use the ORDER BY clause.

If more than one *order-by-expression* is specified (separated by commas), then the first one determines the major sort order.

Integers can be substituted for column names in the ORDER BY clause. For example, if the *order-by-expression* is 2, then the results are ordered by values in the second column. Note that if you use a floating-point value (for example, 2.3) instead of an integer, then FedSQL issues an error message.

In the ORDER BY clause, you can specify any column of a table that is specified in the FROM clause of a query expression, regardless of whether that column has been included in the query's SELECT clause. However, if SELECT DISTINCT is specified, or if the SELECT statement contains a UNION operator, the sort column must appear in the query's SELECT clause.

Note: SAS missing values or null values are treated as the lowest possible values.

Note: For an SPD Engine data set, utility files are used for certain operations that need extra space. The ORDER BY clause requires a utility file and the Base SAS UTILLOC system option allocates space for that utility file. For more information, see *SAS Scalable Performance Data Engine: Reference*.

LIMIT Clause

Description

Specifies the number of rows that the SELECT statement returns.

Syntax

LIMIT {*count* | ALL}

Arguments

count

specifies the number of rows that the SELECT statement returns.

Tip *count* can be an integer or any simple expression that resolves to an integer value.

ALL

specifies that all rows are returned.

Details

The LIMIT clause can be used alone or in conjunction with the OFFSET clause. The OFFSET clause specifies the number of rows to skip before the SELECT statement starts to return rows.

Note: When you use the LIMIT clause, it is recommended that you use an ORDER BY clause to create an ordered sequence. Otherwise you can get an unpredictable subset of a query's rows.

OFFSET Clause

Description

Specifies the number of rows to skip before the SELECT statement starts to return rows.

Syntax

OFFSET *number*

Arguments

number

specifies the number of rows to skip.

Tip *number* can be an integer or any simple expression that resolves to an integer value.

Details

The OFFSET clause can be used alone or in conjunction with the LIMIT clause. The OFFSET clause specifies the number of rows to skip before the SELECT statement starts to return rows.

Note: When you use the OFFSET clause, it is recommended that you use an ORDER BY clause to create an ordered sequence. Otherwise you get an unpredictable subset of a query's rows.

UNION Operator

Descriptions

Combines the result of two or more queries into a single result table.

Syntax

```
{<query-specification> | <query-expression>}
  UNION [ALL | DISTINCT] {<query-specification> | <query-expression>}
```

Arguments

<query-specification> | <query-expression>

specifies one or more SELECT statements that produces a virtual table.

See [“SELECT Statement” on page 386](#)

[“Overview of Subqueries” on page 44](#)

UNION

specifies that multiple result tables are combined and returned as a single result table.

ALL

specifies that all rows, including duplicates, are included in the result table. If not specified, all rows are returned.

DISTINCT

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

See [“DISTINCT Predicate” on page 327](#)

Details

The UNION set operator produces a table that contains all the unique rows that result from both queries. That is, the result table contains rows produced by the first query, the second query, or both.

Columns are appended by position in the tables, regardless of the column names. However, the data type of the corresponding columns must match or the union does not occur. Also, the number and the order of the columns must be identical in all queries.

The names of the columns in the result table are the names of the columns from the first query expression or query-specification unless a column (such as an expression) has no name in the first query expression or query-specification. In such a case, the name of that column in the result table is “column”.

The UNION set operator automatically eliminates duplicate rows from its result tables. The optional ALL keyword preserves the duplicate rows, reduces the execution by one step, and thereby improves the query's performance. You use it when you want to display all the rows resulting from the query, rather than just the unique rows. The ALL keyword is used only when a set operator is also specified.

EXCEPT Operator**Description**

Combines the result of two or more queries into a single result table that contains only rows that are in the first query but not in the second query.

Syntax

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

Arguments

<query-specification> | <query-expression>

specifies one or more SELECT statements that produces a virtual table.

See [“SELECT Statement” on page 386](#)

[“Overview of Subqueries” on page 44](#)

EXCEPT

specifies that multiple result tables are combined and only those rows that are in the first result table and not in the second result table are included.

ALL

specifies that all rows, including duplicates, are included in the result table. If not specified, all rows are returned.

DISTINCT

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

See [“DISTINCT Predicate” on page 327](#)

CORRESPONDING

specifies the columns to include in the query.

Tip If you do not specify the BY clause, the result table will include every column that appears in both of the tables.

BY

specifies that only these columns be included in the result table.

column

specifies the name of the column.

Restriction Every *column* must be a valid column in both tables.

Details

The EXCEPT set operator produces (from the first query) a result table that has unique rows that are not in the second query. If the intermediate result from the first query has at least one occurrence of a row that is not in the intermediate result of the second query, then that row (from the first query) is included in the result table.

The EXCEPT set operator automatically eliminates duplicate rows from its result tables. The optional ALL keyword preserves the duplicate rows, reduces the execution by one step, and thereby improves the query's performance. You use it when you want to display all the rows resulting from the query, rather than just the unique rows. The ALL keyword is used only when a set operator is also specified.

INTERSECT Operator**Description**

Combines the result of two or more queries into a single result table that contains only rows that are common to both queries.

Syntax

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

Arguments**<query-specification> | <query-expression>**

specifies one or more SELECT statements that produces a virtual table.

See [“SELECT Statement” on page 386](#)[“Overview of Subqueries” on page 44](#)**INTERSECT**

specifies that multiple result tables are combined and only those rows that are common to both result tables are included.

ALL

specifies that all rows, including duplicates, are included in the result table. If not specified, all rows are returned.

DISTINCT

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

See [“DISTINCT Predicate” on page 327](#)**CORRESPONDING**

specifies the columns to include in the query.

Tip If you do not specify the BY clause, the result table will include every column that appears in both of the tables.**BY**

specifies that only these columns to be included in the result table.

column

specifies the name of the column.

Restriction Every *column* must be a valid column in both tables.**Details**

The INTERSECT operator produces a result table that has rows that are common to both tables.

The INTERSECT set operator automatically eliminates duplicate rows from its result tables. The optional ALL keyword preserves the duplicate rows, reduces the execution by one step, and thereby improves the query's performance. You use it when you want to display all the rows resulting from the query, rather than just the unique rows. The ALL keyword is used only when a set operator is also specified.

<search-condition>**Description**

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

Syntax

<search-condition>::=

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

<sql-expression>::=

expression {operator | predicate} expression

Arguments**NOT**

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

Table 8.6 Truth Table for the NOT Operator

NOT	Result
<i>True</i>	False
<i>False</i>	True
<i>Unknown</i>	Unknown

AND

combines two conditions by finding observations that satisfy both conditions. This table outlines the outcomes when you compare TRUE and FALSE values using the AND operator.

Table 8.7 Truth Table for the AND Operator

AND	True	False	Unknown
<i>True</i>	True	False	Unknown
<i>False</i>	False	False	False
<i>Unknown</i>	Unknown	False	Unknown

OR

combines two conditions by finding observations that satisfy either condition or both. This table outlines the outcomes when you compare TRUE and FALSE values using the OR operator.

Table 8.8 Truth Table for the OR Operator

OR	True	False	Unknown
<i>True</i>	True	True	True

OR	True	False	Unknown
<i>False</i>	True	False	Unknown
<i>Unknown</i>	True	Unknown	Unknown

<sql-expression>

specifies any valid SQL expression.

See [“<sql-expression>” on page 339](#)

Details

The search condition specifies which rows are returned in a result table for a SELECT statement. Within the SELECT statement, the search condition is used in the WHERE clause, the HAVING clause, and the ON clause with joins.

The order of precedence for the logical operators is NOT, AND, and then OR, but you can override the order by using parentheses. Everything within the parentheses is evaluated first to yield a single value before that value can be used by any operator outside of the parentheses.

There can be precision issues when REAL values are involved in a search condition. To avoid these issues, it is recommended that you define or create a cast for the value in the condition. In other words, wherever the value is used in the original query, force a cast with this syntax.

CAST (*value AS type*)

Here are some examples:

```
select * from test where x = cast (1.0e20 as real);
select cast (1.0e20 as real) from test;
select cast (col1 as real) from test;
```

Note: The search condition is also used with the [“UPDATE Statement” on page 412](#) and the [“DELETE Statement” on page 377](#). For the UPDATE statement, the search specification specifies which rows are updated. For the DELETE statement, the search specification specifies which rows are deleted.

See Also**Concepts:**

- [“FedSQL Pass-Through Facility” on page 59](#)

Statements:

- [“DELETE Statement” on page 377](#)
- [“UPDATE Statement” on page 412](#)

UPDATE Statement

Modifies a column's values in existing rows of a table.

Category: Data Manipulation

Supports: [“EXECUTE Statement” on page 381](#)

Data source: SAS data set, SPD Engine data set, Aster, DB2 under UNIX and PC, Greenplum, MDS, MySQL, Netezza, ODBC, Oracle, PostgreSQL, SAP HANA, Sybase IQ, Teradata

Syntax

UPDATE *table*

```
{
  SET column=<sql-expression> [, ...column=<sql-expression>]
    | column=value-expression [, ...column=value-expression]
}
[WHERE <sql-expression> | value-expression];
```

Arguments

table

specifies a table name.

Restriction A Teradata table must have a primary key defined to be updated with the UPDATE statement.

column

specifies a column name.

<*sql-expression*>

specifies any valid SQL expression.

See [“<sql-expression>” on page 339](#)

value-expression

specifies any valid value expression.

Tip You can use parameter arrays in the INSERT statement.

Details

The UPDATE statement changes the values in all or part of an existing row in a table.

The SET clause specifies which columns to modify and the values to assign them. Columns that are not SET retain their previous values. In the SET clause, a column reference on the left side of the equal sign can also appear as part of the expression on the right side of the equal sign. The SET clause uses the current value of the column in the expression. For example, you could use this expression to give employees a \$1,000 holiday bonus.

```
update payroll set salary = salary + 1000;
```

The WHERE clause enables you to choose which rows to update. If you omit the WHERE clause, then all the rows are updated. When you use a WHERE clause, only the rows that meet the WHERE clause condition are updated.

Note: If row-level permissions are in effect for the table, you cannot update rows in the table. Row-level security is available only with SAS Federation Server. For more information about row-level security, see the SAS Federation Server documentation.

Comparisons

The DELETE statement enables you to delete rows from a table. The INSERT statement enables you to insert new rows into a table. The UPDATE statement enables you to change rows in a table.

See Also

Statements:

- [“DELETE Statement” on page 377](#)
- [“INSERT Statement” on page 382](#)
- WHERE clause in the [“SELECT Statement” on page 386](#)

Chapter 9

FedSQL Statement Table Options

Overview of Statement Table Options	417
About FedSQL Statement Table Options	417
Restrictions	417
How Table Options Interact with Other Types of Options	417
FedSQL Statement Table Option Syntax	418
Understanding BULKLOAD Table Options	418
FedSQL Statement Table Options by Data Source	418
Dictionary	426
ALTER= Table Option	426
ASYNCINDEX= Table Option	427
BL_ALLOW_READ_ACCESS= Table Option	428
BL_COPY_LOCATION= Table Option	428
BL_CPU_PARALLELISM= Table Option	429
BL_DATA_BUFFER_SIZE= Table Option	430
BL_DEFAULT_DIR= Table Option	431
BL_DISK_PARALLELISM= Table Option	431
BL_ERRORS= Table Option	432
BL_EXCEPTION= Table Option	433
BL_INDEXING_MODE= Table Option	434
BL_LOAD= Table Option	434
BL_LOAD_REPLACE= Table Option	435
BL_LOG= Table Option	435
BL_LOGFILE= Table Option	436
BL_OPTIONS= Table Option	437
BL_PARALLEL Table Option	438
BL_PORT_MAX= Table Option	438
BL_PORT_MIN= Table Option	439
BL_RECOVERABLE= Table Option	439
BL_REMOTE_FILE= Table Option	440
BL_SKIP= Table Option	441
BL_SKIP_INDEX_MAINTENANCE= Table Option	441
BL_SKIP_UNUSABLE_INDEXES= Table Option	442
BL_WARNING_COUNT= Table Option	443
BLOCKSIZE= Table Option	443
BUFNO= Table Option	444
BUFSIZE= Table Option	445
BULKLOAD= Table Option	446
BULKOPTS= Table Option	447
COMPRESS= Table Option	448

COPIES= Table Option	450
DBCREATE_INDEX_OPTS=	450
DBCREATE_TABLE_OPTS= Table Option	451
ENCRYPT= Table Option	453
ENCRYPTKEY= Table Option	455
ENDOBS= Table Option	458
EXTENDOBS_COUNTER= Table Option	458
GP_DISTRIBUTED_BY= Table Option	459
HASH= Table Option	460
IDXNAME= Table Option	461
IDXWHERE= Table Option	462
IOBLOCKSIZE= Table Option	463
LABEL= Table Option	464
LOCKTABLE= Table Option	465
ORDERBY= Table Option	466
ORHINTS= Table Option	467
ORNUMERIC= Table Option	467
PADCOMPRESS= Table Option	468
PARTSIZE= Table Option	469
PARTITION= Table Option	471
PARTITION_KEY= Table Option	472
PERM= Table Option	472
POINTOBS= Table Option	473
PW= Table Option	474
READ= Table Option	474
REUSE= Table Option	475
SQUEEZE= Table Option	476
STARTOBS= Table Option	476
TABLE_TYPE=	477
TD_BUFFER_MODE= Table Option	478
TD_CHECKPOINT= Table Option	478
TD_DATA_ENCRYPTION= Table Option	479
TD_DROP_ERROR_TABLE= Table Option	479
TD_DROP_LOG_TABLE= Table Option	480
TD_DROP_WORK_TABLE= Table Option	480
TD_ERROR_LIMIT= Table Option	481
TD_ERROR_TABLE_1= Table Option	481
TD_ERROR_TABLE_2= Table Option	482
TD_LOG_MECH_TYPE= Table Option	483
TD_LOG_MECH_DATA= Table Option	484
TD_LOG_TABLE= Table Option	485
TD_LOGDB= Table Option	485
TD_MAX_SESSIONS= Table Option	486
TD_MIN_SESSIONS= Table Option	487
TD_NOTIFY_LEVEL= Table Option	487
TD_NOTIFY_METHOD= Table Option	488
TD_NOTIFY_STRING= Table Option	489
TD_PACK= Table Option	489
TD_PACK_MAXIMUM= Table Option	490
TD_PAUSE_ACQ= Table Option	491
TD_SESSION_QUERY_BAND= Table Option	491
TD_TENACITY_HOURS= Table Option	492
TD_TENACITY_SLEEP= Table Option	492
TD_TPT_OPER= Table Option	493
TD_TRACE_LEVEL= and TD_TRACE_LEVEL_INF= Table Options	494
TD_TRACE_OUTPUT= Table Option	495

TD_WORK_TABLE= Table Option	496
TD_WORKING_DB= Table Option	496
THREADNUM= Table Option	497
TYPE= Table Option	498
UCA= Table Option	498
UNIQUESAVE= Table Option	499
WHEREINDEX= Table Option	499
WRITE= Table Option	500

Overview of Statement Table Options

About FedSQL Statement Table Options

FedSQL statement table options specify actions that apply only to the table with which they appear. They enable you to perform operations such as specifying a password for a SAS file and renaming columns.

You can specify table options in a FedSQL statement in which you specify a table name, such as the CREATE TABLE, ALTER TABLE, or SELECT statement.

Restrictions

The availability and behavior of FedSQL statement options are data-source specific.

How Table Options Interact with Other Types of Options

Many types of options share the same name and have the same function. For example, you can compress a SAS data set by specifying the COMPRESS= table option as a connection string option and as a FedSQL statement table option.

When more than one type of option with the same function is specified, the software follows the following order of precedence:

1. FedSQL statement table option
2. data source connection string option

For example, if a connection string includes COMPRESS=NO, and a FedSQL statement includes the table option COMPRESS=YES, then the table option overrides the connection string specification for the named table.

If you are submitting FedSQL statements in a Base SAS session, the following order of precedence applies:

1. FedSQL statement option
2. LIBNAME statement option
3. data source connection string option
4. system option

That is, FedSQL statement table options override a LIBNAME statement option, which overrides a data source connection string option.

FedSQL Statement Table Option Syntax

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

CAUTION:

While specifying the syntax for table options, you cannot have a space between the left brace { and the OPTIONS keyword. A space results in a syntax error. For example, this statement is correct:

```
create table temp {options dbcreate_table_opts='primary index(b)'}
(a int, b int);
```

The following statement, however, is incorrect and results in an error message:

```
create table temp { options dbcreate_table_opts='primary index(b)'}
(a int, b int);
```

{OPTIONS *option-1=value* [... *option-n=value*]}

These examples show table options in FedSQL statements:

```
create table salary {options encrypt=yes read=green};
```

```
select * from salary {options read=green};
```

Understanding BULKLOAD Table Options

Table options that are related to bulk loading are available only when the FedSQL language processes the call, as opposed to when the driver uses the native SQL language. The FedSQL language processor is used over native SQL to process requests that create tables or insert data from multiple data sources. In order for the table options to be enforced, the data source that supports them must be specified in the CREATE TABLE or INSERT INTO part of the query. The FedSQL language processor is also used to process CREATE TABLE and INSERT statements for a single data source when the requests specify a SAS function.

FedSQL Statement Table Options by Data Source

The following table lists the table options that are supported in FedSQL programs by the data source that supports them. The options are listed alphabetically for each data source.

Table 9.1 List of Supported Table Options by Data Source

Data Source	Language Element	Category	Description
Aster	“PARTITION_KEY= Table Option” on page 472	Table control	Specifies the column name to use as the partition key for creating fact tables.
DB2 under UNIX and PC	“BL_ALLOW_READ_ACCESS= Table Option” on page 428	Bulk loading	Specifies that the original table data is still visible to readers during bulk load.
	“BL_COPY_LOCATION= Table Option” on page 428	Bulk loading	Specifies the directory to which DB2 saves a copy of the loaded data.
	“BL_CPU_PARALLELISM= Table Option” on page 429	Bulk loading	Specifies the number of processes or threads to use when building table objects.
	“BL_DATA_BUFFER_SIZE= Table Option” on page 430	Bulk loading	Specifies the total amount of memory to allocate for the bulk load utility to use as a buffer for transferring data.
	“BL_DISK_PARALLELISM= Table Option” on page 431	Bulk loading	Specifies the number of processes or threads to use when writing data to disk.
	“BL_EXCEPTION= Table Option” on page 433	Bulk loading	Specifies the exception table into which rows in error are copied.
	“BL_INDEXING_MODE= Table Option” on page 434	Bulk loading	Specifies which scheme the DB2 load utility should use for index maintenance.
	“BL_LOAD_REPLACE= Table Option” on page 435	Bulk loading	Specifies whether DB2 appends or replaces rows during bulk loading.
	“BL_LOG= Table Option” on page 435	Bulk loading	Identifies a log file that contains information such as statistics and error information for a bulk load.
	“BL_OPTIONS= Table Option” on page 437	Bulk loading	Passes options to the DBMS bulk load facility, affecting how it loads and processes data.
	“BL_PORT_MAX= Table Option” on page 438	Bulk loading	Sets the highest available port number for concurrent uploads.
	“BL_PORT_MIN= Table Option” on page 439	Bulk loading	Sets the lowest available port number for concurrent uploads.
	“BL_RECOVERABLE= Table Option” on page 439	Bulk loading	Specifies whether the LOAD process is recoverable.
	“BL_REMOTE_FILE= Table Option” on page 440	Bulk loading	Specifies the base filename and location of DB2 LOAD temporary files.
	“BL_WARNING_COUNT= Table Option” on page 443	Bulk loading	Specifies the maximum number of row warnings to allow before the load fails.

Data Source	Language Element	Category	Description
	“BULKLOAD= Table Option” on page 446	Bulk loading	Loads rows of data as one unit.
	“BULKOPTS= Table Option” on page 447	Bulk loading	Container for bulk load options. This option must follow BULKLOAD=YES.
	“DBCREATE_TABLE_OPTS = Table Option” on page 451	Table control	Specifies DBMS-specific syntax to be added to the CREATE TABLE statement.
Greenplum	“DBCREATE_TABLE_OPTS = Table Option” on page 451	Table control	Specifies DBMS-specific syntax to be added to the CREATE TABLE statement.
	“GP_DISTRIBUTED_BY= Table Option” on page 459	Table control	Specifies the distribution key for the table being created.
MDS	“BULKLOAD= Table Option” on page 446	Bulk loading	Loads rows of data as one unit.
MySQL	“DBCREATE_TABLE_OPTS = Table Option” on page 451	Table control	Specifies DBMS-specific syntax to be added to the CREATE TABLE statement.
Netezza	“DBCREATE_TABLE_OPTS = Table Option” on page 451	Table control	Specifies DBMS-specific syntax to be added to the CREATE TABLE statement.
ODBC	“DBCREATE_INDEX_OPTS =” on page 450.	Index control	Specifies DBMS-specific syntax to be added to the CREATE INDEX statement.
Oracle	“BL_DEFAULT_DIR= Table Option” on page 431	Bulk loading	Specifies where bulk load creates all intermediate files.
	“BL_ERRORS= Table Option” on page 432	Bulk loading	Specifies that after the indicated number of errors is received, that the load should stop.
	“BL_LOAD= Table Option” on page 434	Bulk loading	Specifies that after the indicated number of rows is loaded, that the load should stop.
	“BL_LOGFILE= Table Option” on page 436	Bulk loading	Specifies the filename for the bulk load log file.
	“BL_PARALLEL Table Option” on page 438	Bulk loading	Specifies whether to perform a parallel bulk load.
	“BL_RECOVERABLE= Table Option” on page 439	Bulk loading	Determines whether the LOAD process is recoverable.
	“BL_SKIP= Table Option” on page 441	Bulk loading	Specifies to skip the indicated number of rows before starting the bulk load.
	“BL_SKIP_INDEX_MAINTENANCE= Table Option” on page 441	Bulk loading	Specifies whether to perform index maintenance on the bulk load.

Data Source	Language Element	Category	Description
	“BL_SKIP_UNUSABLE_IND EXES= Table Option” on page 442	Bulk loading	Specifies whether to skip index entries that are in an unusable state and continue with the bulk load.
	“BULKLOAD= Table Option” on page 446	Bulk loading	Loads rows of data as one unit.
	“BULKOPTS= Table Option” on page 447	Bulk loading	Container for bulk load options. This option must follow BULKLOAD=YES.
	“DBCREATE_TABLE_OPTS = Table Option” on page 451.	Table control	Specifies DBMS-specific syntax to be added to the CREATE TABLE statement.
	“ORHINTS= Table Option” on page 467	Data control	Specifies Oracle hints to pass to Oracle from FedSQL.
	“ORNUMERIC= Table Option” on page 467	Table control	Specifies how numbers read from or inserted into the Oracle NUMBER column will be treated.
SAP HANA	“DBCREATE_TABLE_OPTS = Table Option” on page 451	Table control	Specifies DBMS-specific syntax to be added to the CREATE TABLE statement.
	“TABLE_TYPE=” on page 477	Data access	Specifies the type of table storage FedSQL will use when creating tables in SAP HANA.
SAS data set	“ALTER= Table Option ” on page 426	Table control	Assigns an ALTER password to a SAS data set that prevents users from replacing or deleting the file, and enables access to a read- or write-protected file.
	“BUFNO= Table Option ” on page 444	Table control	Specifies the number of buffers to be allocated for processing a SAS data set.
	“BUFSIZE= Table Option ” on page 445	Table control	Specifies the size of a permanent buffer page for an output SAS data set.
	“COMPRESS= Table Option ” on page 448	Table control	Specifies how rows are compressed in a new output data set.
	“ENCRYPT= Table Option ” on page 453	Table control	Specifies whether to encrypt an output SAS data set.
	“ENCRYPTKEY= Table Option” on page 455.	Table control	Specifies a key value for AES encryption.
	“EXTENDOBSCOUNTER= Table Option” on page 458	Table control	Specifies whether to extend the maximum observation count in a new output SAS data file.
	“IDXNAME= Table Option ” on page 461	User control of index usage	Directs SAS to use a specific index to match the conditions of a WHERE clause.

Data Source	Language Element	Category	Description
	“IDXWHERE= Table Option” on page 462	User control of index usage	Specifies whether SAS uses an index search or a sequential search to match the conditions of a WHERE clause.
	“LABEL= Table Option” on page 464	Observation control	Specifies a label for a SAS data set.
	“LOCKTABLE= Table Option” on page 465	Table control	Places shared or exclusive locks on tables.
	“POINTOBS= Table Option” on page 473	Table control	Specifies whether SAS creates compressed data sets whose observations can be randomly accessed or sequentially accessed.
	“PW= Table Option ” on page 474	Table control	Assigns a READ, WRITE, and ALTER password to a SAS data set, and enables access to a password-protected file.
	“READ= Table Option ” on page 474	Table control	Assigns a READ password to a SAS data set that prevents users from reading the file, unless they enter the password.
	“REUSE= Table Option” on page 475	Table control	Specifies whether new rows can be written to freed space in a compressed SAS data set.
	“TYPE= Table Option ” on page 498	Table control	Specifies the data set type for a specially structured SAS data set.
	“WRITE= Table Option ” on page 500	Table control	Assigns a WRITE password to a SAS data set or an SPD data set that prevents users from writing to the file or that enables access to a write-protected file.
SASHDAT file	“BLOCKSIZE= Table Option” on page 443	Table control	Specifies the size of a SASHDAT file block in bytes, kilobytes, megabytes, or gigabytes.
	“COPIES= Table Option” on page 450	Table control	Specifies how many copies are made when file blocks are written to HDFS.
	“HASH= Table Option” on page 460	Table control	Indicates that the distribution properties of the partitions depends on a hash function.
	“LABEL= Table Option” on page 464	Table control	Specifies a label for a SASHDAT file.
	“ORDERBY= Table Option” on page 466	Table control	Specifies the columns by which to order the data within a partition.
	“PARTITION= Table Option” on page 471	Table control	Specifies the list of columns to use to partition the SASHDAT file.

Data Source	Language Element	Category	Description
	“PERM= Table Option” on page 472	Security	Specifies the permission setting when writing a SASHDAT file to HDFS.
	“SQUEEZE= Table Option” on page 476	File control	Specifies to write the SASHDAT file in a compressed format.
	“UCA= Table Option” on page 498	Table control	Specifies to use UCA collation to order character variables in the ORDERBY= option.
SPD Engine data set	“ALTER= Table Option ” on page 426	Table control	Assigns an ALTER password to an SPD Engine data set that prevents users from replacing or deleting the file, and enables access to a read- or write-protected file.
	“ASYNINDEX= Table Option” on page 427	User control of index usage	Specifies to create indexes in parallel when creating multiple indexes on an SPD Engine data set.
	“COMPRESS= Table Option ” on page 448	Table control	Specifies to compress SPD Engine data sets on disk as they are being created.
	“ENCRYPT= Table Option ” on page 453	Table control	Specifies whether to encrypt an output SPD Engine data set.
	“ENCRYPTKEY= Table Option” on page 455	Table control	Specifies a key value for AES encryption.
	“ENDOBS= Table Option” on page 458	Observation control	Specifies the end observation number in a user-defined range of observations to be processed.
	“IDXWHERE= Table Option” on page 462	User control of index usage	Specifies to use indexes when processing WHERE expressions in the SPD Engine.
	“IOBLOCKSIZE= Table Option” on page 463	Table control	Specifies the size in bytes of a block of observations to be used in an I/O operation.
	“LABEL= Table Option” on page 464	Observation control	Specifies a label for an SPD Engine data set.
	“PADCOMPRESS= Table Option” on page 468	Table control	Specifies the number of bytes to add to compressed blocks in a data set opened for OUTPUT or UPDATE.
	“PARTSIZE= Table Option” on page 469	Table control	Specifies the size of the data component partitions in an SPD Engine data set.
	“PW= Table Option ” on page 474	Table control	Assigns a READ, WRITE, and ALTER password to a SAS data set, and enables access to a password-protected file.

Data Source	Language Element	Category	Description
	“READ= Table Option ” on page 474	Table control	Assigns a READ password to a SAS data set that prevents users from reading the file, unless they enter the password.
	“STARTOBS= Table Option” on page 476.	Observation control	Specifies the starting observation number in a user-defined range of observations to be processed.
	“THREADNUM= Table Option” on page 497	Table control	Specifies the maximum number of I/O threads the SPD Engine can spawn for processing an SPD Engine data set.
	“TYPE= Table Option ” on page 498.	Table control	Specifies the data set type for a specially structured SPD Engine data set.
	“UNIQUESAVE= Table Option” on page 499	User control of index usage	Specifies to save observations with nonunique key values (the rejected observations) to a separate data set when appending or inserting observations to data sets with unique indexes.
	“WHEREINDEX= Table Option” on page 499	User control of index usage	Specifies a list of indexes to exclude when making WHERE expression evaluations.
	“WRITE= Table Option ” on page 500	Table control	Assigns a WRITE password to an SPD Engine data set that prevents users from writing to the file or that enables access to a write-protected file.
Teradata	“BULKLOAD= Table Option” on page 446.	Bulk loading	Loads rows of data as one unit.
	“BULKOPTS= Table Option” on page 447.	Bulk loading	Container for bulk load options. This option must follow BULKLOAD=YES.
	“DBCREATE_TABLE_OPTS = Table Option” on page 451	Table control	Specifies DBMS-specific syntax to be added to the CREATE TABLE statement.
	“TD_BUFFER_MODE= Table Option” on page 478.	Bulk loading	Specifies whether the LOAD method is used.
	“TD_CHECKPOINT= Table Option” on page 478	Bulk loading	Specifies when the TPT operation issues a checkpoint or savepoint to the database.
	“TD_DATA_ENCRYPTION= Table Option” on page 479	Bulk loading	Activates data encryption.
	“TD_DROP_LOG_TABLE= Table Option” on page 480	Bulk loading	Drops the log table at the end of the job, whether the job completed successfully or not.

Data Source	Language Element	Category	Description
	“TD_DROP_ERROR_TABLE = Table Option” on page 479	Bulk loading	Drops the error tables at the end of the job, whether the job completed successfully or not.
	“TD_DROP_WORK_TABLE= Table Option” on page 480	Bulk loading	Drops the work table at the end of the job, whether the job completed successfully or not.
	“TD_ERROR_LIMIT= Table Option” on page 481	Bulk loading	Specifies the maximum number of records that can be stored in an error table.
	“TD_ERROR_TABLE_1= Table Option” on page 481	Bulk loading	Specifies a name for the first error table.
	“TD_ERROR_TABLE_2= Table Option” on page 482	Bulk loading	Specifies a name for the second error table.
	“TD_LOG_TABLE= Table Option” on page 485	Bulk loading	Specifies the name of the restart log table.
	“TD_LOG_MECH_TYPE= Table Option” on page 483	Bulk loading	Specifies the logon mechanism for a bulk load.
	“TD_LOG_MECH_DATA= Table Option” on page 484	Bulk loading	Specifies additional data for the logon mechanism.
	“TD_LOGDB= Table Option” on page 485	Bulk loading	Specifies the database where the TPT utility tables are created.
	“TD_MAX_SESSIONS= Table Option” on page 486	Bulk loading	Specifies the maximum number of logon sessions that TPT can acquire for a job.
	“TD_MIN_SESSIONS= Table Option” on page 487	Bulk loading	Specifies the minimum number of sessions for TPT to acquire before a job starts.
	“TD_NOTIFY_LEVEL= Table Option” on page 487	Bulk loading	Specifies the level at which log events are recorded.
	“TD_NOTIFY_METHOD= Table Option” on page 488	Bulk loading	Specifies the method for reporting events.
	“TD_NOTIFY_STRING= Table Option” on page 489.	Bulk loading	Defines a string that precedes all messages sent to the system log.
	“TD_PACK= Table Option” on page 489	Bulk loading	Specifies the number of statements to pack into a multistatement request.
	“TD_PACK_MAXIMUM= Table Option” on page 490	Bulk loading	Enables the Stream operator to determine the maximum possible pack factor for the current Stream job.

Data Source	Language Element	Category	Description
	“TD_PAUSE_ACQ= Table Option” on page 491	Bulk loading	Forces a pause between the acquisition phase and the application phase of a load job.
	“TD_SESSION_QUERY_BA ND= Table Option” on page 491	Bulk loading	Passes a string of user-specified name=value pairs for use by the TPT session.
	“TD_TENACITY_HOURS= Table Option” on page 492	Bulk loading	Specifies the amount of time the TPT operator continues trying to log on to the Teradata database.
	“TD_TENACITY_SLEEP= Table Option” on page 492.	Bulk loading	Specifies the amount of time the TPT operator pauses, before retrying to log on to the Teradata database.
	“TD_TPT_OPER= Table Option” on page 493	Bulk loading	Specifies the load operator used by the Teradata Parallel Transporter.
	“TD_TRACE_LEVEL= and TD_TRACE_LEVEL_INF= Table Options” on page 494	Bulk loading	Specify the trace levels for driver tracing. TD_TRACE_LEVEL sets the primary trace level. TD_TRACE_LEVEL_INF sets the secondary trace level.
	“TD_TRACE_OUTPUT= Table Option” on page 495.	Bulk loading	Specifies the name of the external file used for trace messages.
	“TD_WORKING_DB= Table Option” on page 496	Bulk loading	Specifies the database where the table is to be created.
	“TD_WORK_TABLE= Table Option” on page 496	Bulk loading	Specifies a name for the TPT work table.

Dictionary

ALTER= Table Option

Assigns an ALTER password to a data set that prevents users from replacing or deleting the file, and enables access to a Read- or Write-protected file.

Category: Table Control

Data source: SAS data set, SPD Engine data set

Note: Check your log after this operation to ensure that the password values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in *SAS Language Reference: Concepts*.

Syntax

ALTER= *alter-password*

Arguments

alter-password

specifies a password.

Restriction *alter-password* must be a valid SAS name.

Details

The ALTER= table option applies only to a SAS data set or an SPD Engine data set. You can use this option to assign a password or to access a Read-protected, Write-protected, or alter-protected file. When you replace a data set that is protected with an ALTER password, the new data set inherits the ALTER password.

Note: A SAS password does not control access to a SAS file beyond the SAS system. You should use the operating system-supplied utilities and file-system security controls to control access to SAS files outside of SAS.

ASYNCINDEX= Table Option

Specifies to create indexes in parallel when creating multiple indexes on an SPD Engine data set.

Valid in: CREATE INDEX Statement

Category: User Control of Index Usage

Data source: SPD Engine data set

Syntax

ASYNCINDEX= YES | NO

Arguments

YES

creates the indexes in parallel (asynchronously).

NO

creates one index at a time (synchronously). This is the default value.

Details

The SPD Engine can create multiple indexes for a data set at the same time. The SPD Engine spawns a single thread for each index created, and then processes the threads simultaneously. Although creating indexes in parallel is much faster than creating one index at a time, the default for this option is NO. Parallel creation requires additional utility work space and additional memory, which might not be available. If the index creation fails due to insufficient resources, you can do one of the following:

- set the SAS system option to MEMSIZE=0
- increase the size of the utility file space using the SPDEUTILLOC= system option

You increase the memory space that is used for index sorting using the SPDEINDEXSORTSIZE= system option. If you specify to create indexes in parallel, specify a large-enough space using the SPDEUTILLOC= system option.

BL_ALLOW_READ_ACCESS= Table Option

Specifies that the original table data is still visible to readers during bulk load.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified within the “BULKOPTS= Table Option” on page 447
Data source:	DB2 under UNIX and PC

Syntax

BL_ALLOW_READ_ACCESS= YES | NO

Arguments

YES

specifies that the original (unchanged) data in the table is still visible to readers while bulk load is in progress.

NO

specifies that readers cannot view the original data in the table while bulk load is in progress. This is the default value.

Details

For more information about using this option, see the SQLU_ALLOW_READ_ACCESS parameter in *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)

BL_COPY_LOCATION= Table Option

Specifies the directory to which DB2 saves a copy of the loaded data.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified within the “BULKOPTS= Table Option” on page 447
Data source:	DB2 under UNIX and PC

Syntax

BL_COPY_LOCATION= *pathname*

Arguments

pathname

specifies the path where the loaded data is copied.

See Also

Table Options:

- “BL_RECOVERABLE= Table Option” on page 439
- “BULKLOAD= Table Option” on page 446

BL_CPU_PARALLELISM= Table Option

Specifies the number of processes or threads to use when building table objects.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified within the “BULKOPTS= Table Option” on page 447

Data source: DB2 under UNIX and PC

Syntax

BL_CPU_PARALLELISM= *number-of-processes-or-threads*

Arguments

number-of-processes-or-threads

specifies the number of processes or threads that the load utility uses to parse, convert, and format data records when building table objects.

Details

This option exploits intrapartition parallelism and significantly improves load performance. It is particularly useful when loading presorted data, because record order in the source data is preserved.

The maximum number that is allowed is 30. If the value is 0 or has not been specified, the load utility selects an intelligent default. This default is based on the number of available CPUs on the system at run time. If there is insufficient memory to support the specified value, the utility adjusts the value.

When BL_CPU_PARALLELISM is greater than 1, the flushing operations are asynchronous, permitting the loader to exploit the CPU. If tables include either LOB or LONG VARCHAR data, parallelism is not supported. The value is set to 1, regardless of the number of system CPUs or the specified value.

Although use of this parameter is not restricted to symmetric multiprocessor (SMP) hardware, you might not obtain any discernible performance benefit from using it in non-SMP environments.

For more information about using `BL_CPU_PARALLELISM=`, see the `CPU_PARALLELISM` parameter in *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

See Also

Table Options:

- “[BL_DATA_BUFFER_SIZE= Table Option](#)” on page 430
- “[BL_DISK_PARALLELISM= Table Option](#)” on page 431
- “[BULKLOAD= Table Option](#)” on page 446

BL_DATA_BUFFER_SIZE= Table Option

Specifies the total amount of memory to allocate for the bulk load utility to use as a buffer for transferring data.

Category: Bulk Loading

Requirements: Must follow `BULKLOAD=YES`
Must be specified within the “[BULKOPTS= Table Option](#)” on page 447

Data source: DB2 under UNIX and PC

Syntax

`BL_DATA_BUFFER_SIZE= buffer-size`

Arguments

buffer-size

specifies the total amount of memory (in 4KB pages) that is allocated for the bulk load utility to use as buffered space for transferring data within the utility. This setting does not consider the degree of parallelism that is available.

Details

If you specify a value that is less than the algorithmic minimum, the minimum required resource is used and no warning is returned. This memory is allocated directly from the utility heap, the size of which you can modify through the `util_heap_sz` database configuration parameter. If you do not specify a value, the utility calculates an intelligent default at run time. The calculation is based on a percentage of the free space that is available in the utility heap at the time of instantiation of the loader, as well as some characteristics of the table.

It is recommended that the buffer be several extents in size. An *extent* is the unit of movement for data within DB2, and the extent size can be one or more 4KB pages.

The `DATA BUFFER` parameter is useful when you are working with large objects (LOBs) because it reduces input and output waiting time. The data buffer is allocated from the utility heap. Depending on the amount of storage that is available on your system, you should consider allocating more memory for use by the DB2 utilities. You can modify the database configuration parameter `util_heap_sz` accordingly. The default value for the Utility Heap Size configuration parameter is 5,000 4KB pages. Because

load is one of several utilities that use memory from the utility heap, it is recommended that no more than 50% of the pages that are defined by this parameter be made available for the load utility.

For more information about using this option, see the DATA BUFFER parameter in *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

See Also

Table Options:

- “BL_CPU_PARALLELISM= Table Option” on page 429.
- “BL_DISK_PARALLELISM= Table Option” on page 431
- “BULKLOAD= Table Option” on page 446

BL_DEFAULT_DIR= Table Option

Specifies the location where bulk load creates all intermediate files.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified within the “BULKOPTS= Table Option” on page 447

Data source: Oracle

Syntax

BL_DEFAULT_DIR= *host-specific-directory-path*

Arguments

host-specific-directory-path

specifies the host-specific directory path where intermediate bulk-load files are created. The default is the current directory.

Details

The value that you specify for this option is prepended to the filename. Be sure to provide the complete, host-specific directory path, including the file and directory separator character to accommodate all platforms.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 446

BL_DISK_PARALLELISM= Table Option

Specifies the number of processes or threads to use when writing data to disk.

Category: Bulk Loading**Requirements:** Must follow BULKLOAD=YES
Must be specified within the “[BULKOPTS= Table Option](#)” on page 447**Data source:** DB2 under UNIX and PC

Syntax

BL_DISK_PARALLELISM= *number-of-processes-or-threads*

Arguments

number-of-processes-or-threads

specifies the number of processes or threads that the load utility uses to write data records to the table-space containers.

Details

This option exploits the available containers when it loads data and significantly improves load performance.

The maximum number that is allowed is the greater of four times the BL_CPU_PARALLELISM value, which the load utility actually uses, or 50. By default, BL_DISK_PARALLELISM is equal to the sum of the table-space containers on all table spaces that contain objects for the table that is being loaded except where this value exceeds the maximum number that is allowed.

If you do not specify a value, the utility selects an intelligent default that is based on the number of table-space containers and the characteristics of the table.

For more information about using this option, see the DISK_PARALLELISM parameter in *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

See Also

Table Options:

- “[BL_CPU_PARALLELISM= Table Option](#)” on page 429
- “[BULKLOAD= Table Option](#)” on page 446

BL_ERRORS= Table Option

Specifies that, after the indicated number of errors is received, the load should stop.

Category: Bulk Loading**Requirements:** Must follow BULKLOAD=YES
Must be specified within the “[BULKOPTS= Table Option](#)” on page 447**Data source:** Oracle

Syntax

BL_ERRORS= *number*

Arguments

number

specifies the number of errors that should be received before the load stops. The default is 1000000 errors.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 446

BL_EXCEPTION= Table Option

Specifies the exception table into which rows in error are copied.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified within the “BULKOPTS= Table Option” on page 447

Data source: DB2 under UNIX and PC

Syntax

BL_EXCEPTION= *exception-table-name*

Arguments

exception-table-name

specifies the exception table into which rows in error are copied.

Details

Any row that is in violation of a unique index or a primary key index is copied. DATALINK exceptions are also captured in the exception table. If you specify an unqualified table name, the table is qualified with the CURRENT SCHEMA. Information that is written to the exception table is not written to the dump file. In a partitioned database environment, you must define an exception table for those partitions on which the loading table is defined. However, the dump file contains rows that cannot be loaded because they are not valid or contain syntax errors.

For more information about using this option, see the FOR EXCEPTION parameter in *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*. For more information about the load exception table, see the load exception table topics in *IBM DB2 Universal Database Data Movement Utilities Guide and Reference* and *IBM DB2 Universal Database SQL Reference, Volume 1*.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 446

BL_INDEXING_MODE= Table Option

Specifies which scheme the DB2 load utility should use for index maintenance.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified within the “BULKOPTS= Table Option” on page 447
Data source:	DB2 under UNIX and PC

Syntax

BL_INDEXING_MODE= [AUTOSELECT](#) | [REBUILD](#) | [INCREMENTAL](#) | [DEFERRED](#)

Arguments

AUTOSELECT

specifies that the load utility automatically decides between REBUILD or INCREMENTAL mode.

REBUILD

specifies that all indexes are rebuilt.

INCREMENTAL

specifies that indexes are extended with new data.

DEFERRED

specifies that the load utility does not attempt index creation. Indexes are marked as needing a refresh.

Details

For more information about using the values for this option, see *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)

BL_LOAD= Table Option

Specifies that, after the indicated number of rows is loaded, the load should stop.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified within the “BULKOPTS= Table Option” on page 447
Data source:	Oracle

Syntax

BL_LOAD= *number-of-rows*

Arguments

number-of-rows

specifies the number of rows that should be loaded. The first rows from the table will be loaded. The default is to load all rows.

See Also

Table Options:

- “[BULKLOAD= Table Option](#)” on page 446

BL_LOAD_REPLACE= Table Option

Specifies whether DB2 appends or replaces rows during bulk loading.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified within the “[BULKOPTS= Table Option](#)” on page 447

Data source: DB2 under UNIX and PC

Syntax

BL_LOAD_REPLACE= [NO](#) | [YES](#)

Arguments

NO

specifies that the CLI LOAD interface appends new rows of data to the DB2 table. This is the default value.

YES

specifies that the CLI LOAD interface replaces the existing data in the table.

See Also

Table Options:

- “[BULKLOAD= Table Option](#)” on page 446

BL_LOG= Table Option

Identifies a log file that contains information such as statistics and error information for a bulk load.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES

Must be specified within the [“BULKOPTS= Table Option” on page 447](#)

Data source: DB2 under UNIX and PC, Oracle

Syntax

BL_LOG= *"path-and-log-file-name"*

Arguments

path-and-log-file-name

is a file to which information about the loading process is written. The default path and log filename is DBMS-specific.

Details

When the DBMS bulk load facility is invoked, it creates a log file. The contents of the log file are DBMS-specific. The BL_ prefix distinguishes this log file from the one created by the SAS log. If the BL_LOG= table option is specified with the same path and filename as an existing log, the new log replaces the existing log.

If the BL_LOG= table option is not specified, the log file is deleted automatically after a successful operation.

Example

The BL_LOG= table option is specified within the BL_BULKOPTS= table option:

```
bulkload=yes; bulkopts=(bl_log="c:\temp\bulkload.log");
```

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)

BL_LOGFILE= Table Option

Specifies the filename for the bulk load log file.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES

Must be specified within the [“BULKOPTS= Table Option” on page 447](#)

Data source: Oracle

Syntax

BL_LOGFILE= *'log-file-name'*

Arguments

log-file-name

specifies a name for the bulk load log file. The default is a generated filename that has the template `BL_TablenameUniquenumber`.

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)

BL_OPTIONS= Table Option

Passes options to the DBMS bulk load facility, affecting how it loads and processes data.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified within the [“BULKOPTS= Table Option” on page 447](#)

Data source: DB2 under UNIX and PC

Syntax

`BL_OPTIONS= 'option [, ...option]'`

Arguments

option

specifies a valid DB2 option. By default, no options are passed.

Details

The `BL_OPTIONS=` table option enables you to pass options to the DBMS bulk load facility when it is invoked, thereby affecting how data is loaded and processed. You must separate multiple options with commas and enclose the entire string of options in single quotation marks.

This option passes DB2 file type modifiers to DB2 LOAD or IMPORT commands to affect how data is loaded and processed. Not all DB2 file type modifiers are appropriate for all situations. You can specify one or more DB2 file type modifiers with .IXF files. For a list of file type modifiers, see the description of the LOAD and IMPORT utilities in *DB2 Data Movement Utilities Guide and Reference*.

Example

This option is specified within the `BULKOPTS=` table option:

```
bulkload=yes; bulkopts=(bl_options='option1, option2');
```

See Also

Table Options:

- “BULKLOAD= Table Option” on page 446
- “BULKOPTS= Table Option” on page 447

BL_PARALLEL Table Option

Specifies whether to perform a parallel bulk load.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified within the “BULKOPTS= Table Option” on page 447
Data source:	Oracle

Syntax

BL_PARALLEL= YES | NO

Arguments

YES

specifies that a parallel load should be performed.

NO

specifies that a parallel load is not performed. That is the default behavior.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 446

BL_PORT_MAX= Table Option

Sets the highest available port number for concurrent uploads.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified within the “BULKOPTS= Table Option” on page 447
Data source:	DB2 under UNIX and PC

Syntax

BL_PORT_MAX= *integer*

Arguments

integer

specifies a positive integer that represents the highest available port number for concurrent uploads.

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)

BL_PORT_MIN= Table Option

Sets the lowest available port number for concurrent uploads.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified within the “BULKOPTS= Table Option” on page 447
Data source:	DB2 under UNIX and PC

Syntax

BL_PORT_MIN= *integer*

Arguments

integer

specifies a positive integer that represents the lowest available port number for concurrent uploads.

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)

BL_RECOVERABLE= Table Option

Specifies whether the LOAD process is recoverable.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified within the “BULKOPTS= Table Option” on page 447
Data source:	DB2 under UNIX and PC, Oracle

Syntax

BL_RECOVERABLE= YES | NO

Arguments

YES

specifies that the LOAD process is recoverable. For DB2, YES also specifies that BL_COPY_LOCATION= should specify the copy location for the data.

NO

specifies that the LOAD process is not recoverable.

Details

DB2 under UNIX and PC Hosts: The default is NO.

Oracle: The default is YES. Set this option to NO to improve direct load performance. Specifying NO adds the UNRECOVERABLE keyword before the LOAD keyword in the control file.

CAUTION:

Be aware that an unrecoverable load does not log loaded data into the redo log file. Therefore, media recovery is disabled for the loaded table. For more information about the implications of using the UNRECOVERABLE parameter in Oracle, see your Oracle utilities documentation.

See Also

Table Options:

- “BL_COPY_LOCATION= Table Option” on page 428
- “BULKLOAD= Table Option” on page 446

BL_REMOTE_FILE= Table Option

Specifies the base filename and location of DB2 LOAD temporary files.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified within the “BULKOPTS= Table Option” on page 447

Data source: DB2 under UNIX and PC

Syntax

BL_REMOTE_FILE= *pathname-and-base-filename*

Arguments

pathname-and-base-filename

specifies the full pathname and base filename to which DB2 appends extensions (such as .log, .msg and .dat files) to create temporary files during load operations. By default, BL_<table>_<unique-ID> is the form of the base filename.

table

specifies the table name.

unique-ID

specifies a number that prevents collisions in the event of two or more simultaneous bulk loads of a particular table. The table driver generates this number.

Details

When you specify this option, the DB2 LOAD command is used (instead of the IMPORT command).

For *pathname*, specify a location on a DB2 server that is accessed exclusively by a single DB2 server instance, and for which the instance owner has Read and Write permissions. Make sure that each LOAD command is associated with a unique *pathname-and-base-filename* value.

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)

BL_SKIP= Table Option

Specifies to skip the indicated number of rows before starting the bulk load.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified within the “BULKOPTS= Table Option” on page 447
Data source:	Oracle

Syntax

BL_SKIP= *number-of-rows*

Arguments

number-of-rows

specifies the number of rows to skip before beginning the load. The default is 0, which means that no records are skipped.

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)

BL_SKIP_INDEX_MAINTENANCE= Table Option

Specifies whether to perform index maintenance on the bulk load.

Category:	Bulk Loading
------------------	--------------

Requirements:	Must follow BULKLOAD=YES Must be specified within the “ BULKOPTS= Table Option ” on page 447
Data source:	Oracle

Syntax

BL_SKIP_INDEX_MAINTENANCE= [YES](#) | [NO](#)

Arguments

YES

specifies to stop index maintenance on the load. This causes the index partitions that would have had index keys added to them to be marked Index Unusable. The index segment is inconsistent with the data it indexes. Index segments that are not affected by the load retain the Index Unusable state that they had prior to the load.

NO

specifies that index maintenance is performed on the load. This is the default value.

See Also

Table Options:

- “[BULKLOAD= Table Option](#)” on page 446

BL_SKIP_UNUSABLE_INDEXES= Table Option

Specifies whether to skip index entries that are in an unusable state and continue with the bulk load.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified within the “ BULKOPTS= Table Option ” on page 447
Data source:	Oracle

Syntax

BL_SKIP_UNUSABLE_INDEXES= [YES](#) | [NO](#)

Arguments

YES

specifies that the unusable index entry should be skipped. This is the default value.

NO

specifies that the unusable index entry should not be skipped.

Details

If an index in an Index Unusable state is encountered, by default, it is skipped and the load operation continues. This allows the SQL*Loader to load a table with indexes that are in an unusable state before beginning the load. Indexes that are not in an unusable

state at load time are maintained by the SQL*Loader. Indexes that are in an unusable state at load time are not maintained and remain in an unusable state at load completion.

If this bulk load option is not specified, the default value is specified in the Oracle database configuration parameter, SKIP_UNUSABLE_INDEXES. This value is specified in the initialization parameter file. The BL_SKIP_UNUSABLE_INDEXES bulk load table option overrides the value of the SKIP_UNUSABLE_INDEXES configuration parameter in the initialization parameter file.

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)

BL_WARNING_COUNT= Table Option

Specifies the maximum number of row warnings to allow before the load fails.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified within the [“BULKOPTS= Table Option” on page 447](#)

Data source: DB2 under UNIX and PC

Syntax

BL_WARNING_COUNT= *number-of-warnings*

Arguments

number-of-warnings

specifies the maximum number of row warnings to allow before the load fails.

Details

To specify this option, you must first set BULKLOAD=YES and also specify a value for BL_REMOTE_FILE=.

Use this option to limit the maximum number of rows that generate warnings. See the log file for information about why the rows generated warnings.

See Also

Table Options:

- [“BL_REMOTE_FILE= Table Option” on page 440](#)
- [“BULKLOAD= Table Option” on page 446](#)

BLOCKSIZE= Table Option

Specifies the size of a SASHDAT file block in bytes, kilobytes, megabytes, or gigabytes.

Category: Table Control**Default:** 2M, or the length of the record, whichever is greater**Data source:** SASHDAT file

Syntax

BLOCKSIZE=*n*[B] | *n*K | *n*M | *n*G

Arguments

n

specifies the block size to use in bytes. This is the default value when you do not specify a multiplier. If used, the B must be uppercase.

***n*K**

specifies the block size to use in kilobytes. K is a multiplier and must be specified as uppercase.

***n*M**

specifies the block size to use in megabytes. M is a multiplier and must be specified as uppercase.

***n*G**

specifies the block size to use in gigabytes. G is a multiplier and must be specified as uppercase.

Details

You do not have to specify a block size when partitioning data into HDFS. If you do specify a block size, it is considered an upper bound for the partition request.

The actual block size is slightly larger than the value that you specify. This occurs for any of the following reasons:

- to reach the record length.
- to include a metadata header in HDFS for the SASHDAT file.
- to align on a 512-byte boundary.

Example

```
create table hdat.schema1.rent{OPTIONS blocksize=10000}
create table hdat.schema1.rent{OPTIONS blocksize=1000 K}
create table hdat.schema1.rent{OPTIONS blocksize=1G}
```

BUFNO= Table Option

Specifies the number of buffers to be allocated for processing a SAS data set.

Category: Table Control**Data source:** SAS data set

Syntax

BUFNO= *n* | *nK* | *hexX* | MIN | MAX

Arguments

n* | *nK

specifies the number of buffers in multiples of 1 (bytes); 1,024 (kilobytes). For example, a value of **8** specifies 8 buffers, and a value of **1K** specifies 1024 buffers.

Requirement *K* must be uppercased.

hexX

specifies the number of buffers as a hexadecimal value. You must specify the value beginning with a number (0–9), followed by an X. For example, the value **2dX** sets the number of buffers to 45 buffers.

Requirement *X* must be uppercased.

MIN

sets the minimum number of buffers to 0, which causes SAS to use the minimum optimal value for the operating environment. This is the default value.

MAX

sets the number of buffers to the maximum possible number in your operating environment, up to the largest four-byte, signed integer. The largest four-byte, signed integer is $2^{31}-1$, or approximately 2 billion.

Details

The buffer number is not a permanent attribute of the data set; it is valid only for the current operation.

The BUFNO= table option applies to SAS data sets that are opened for input, output, or update.

A larger number of buffers can speed up execution time by limiting the number of input and output (I/O) operations that are required for a particular SAS data set. However, the improvement in execution time comes at the expense of increased memory consumption.

To reduce I/O operations on a small data set as well as speed execution time, allocate one buffer for each page of data to be processed. This technique is most effective if you read the same observations several times during processing.

BUFSIZE= Table Option

Specifies the size of a permanent buffer page for an output SAS data set.

Category: Table Control

Restriction: Use with output data sets only.

Data source: SAS data set

Syntax

BUFSIZE= *n* | *nK* | *nM* | *nG* | *hexX* | MIN | MAX

Arguments

n | *nK* | *nM* | *nG*

specifies the page size in multiples of 1 (bytes); 1,024 (kilobytes); 1,048,576 (megabytes); or 1,073,741,824 (gigabytes). For example, a value of **8** specifies a page size of 8 bytes, and a value of **4k** specifies a page size of 4096 bytes.

Requirement *K*, *M*, and *G* must be uppercased.

hexX

specifies the page size as a hexadecimal value. You must specify the value beginning with a number (0–9), followed by an X. For example, the value **2dx** sets the page size to 45 bytes.

Requirement *X* must be uppercased.

MIN

sets the minimum number of buffers to 0, which causes SAS to use the minimum optimal value for the operating environment.

MAX

sets the page size to the maximum possible number in your operating environment, up to the largest four-byte, signed integer. The largest four-byte, signed integer is $2^{31}-1$, or approximately 2 billion bytes.

Details

The page size is the amount of data that can be transferred for a single I/O operation to one buffer. The page size is a permanent attribute of the data set and is used when the data set is processed.

A larger page size can speed up execution time by reducing the number of times SAS has to read from or write to the storage medium. However, the improvement in execution time comes at the cost of increased memory consumption.

To change the page size, copy the data set and either specify a new page or use the SAS default. To reset the page size to the default value in your operating environment, specify BUFSIZE=0.

Operating Environment Information

The default value for the BUFSIZE= table option is determined by your operating environment and is set to optimize sequential access. To improve performance for direct (random) access, you should change the value for BUFSIZE=.

BULKLOAD= Table Option

Loads rows of data as one unit.

Category: Bulk Loading

Data source: DB2 under UNIX and PC, MDS, Oracle, Teradata

Syntax

BULKLOAD= YES | NO

Arguments

YES

calls a DBMS-specific bulk load facility in order to insert or append rows to a DBMS table.

NO

does not call the DBMS-specific bulk load facility. This is the default value.

Details

Using BULKLOAD=YES is the fastest way to insert rows into a DBMS table.

You can specify additional options in the BULKOPTS= container option (BL_BULKOPTS= for Oracle). The additional options require the BULKOPTS= container option. For more information, see [“BULKOPTS= Table Option” on page 447](#).

MDS: BULKLOAD= is available for insert operations only. When the BULKLOAD= table option is set, newly inserted rows are committed immediately and become visible to existing transactions. When the BULKLOAD= table option is not set, newly inserted rows are not visible until the existing transactions are committed or rolled back.

Teradata: When the BULKLOAD= table option is not set, a simple multi-row insert SQL scheme is used to insert data rows. Specifying BULKLOAD=YES invokes the Teradata Parallel Transporter (TPT) API protocol driver. The default TPT operator is the Stream operator. Use the BULKOPTS= table option to specify a different TPT operator.

See Also

Table Options:

- [“BULKOPTS= Table Option” on page 447](#)

BULKOPTS= Table Option

Provides a container for bulk load options.

Category: Bulk Loading

Alias: In Oracle, this option has the name BL_BULKOPTS=

Requirement: Must follow BULKLOAD=YES

Data source: DB2 under UNIX and PC, Oracle, Teradata

Syntax

BULKOPTS= (*option*[...*option*])

Arguments

option[...option]

specifies one or more bulk load table options separated by spaces.

Details

To specify BULKOPTS= option, you must first specify BULKLOAD=YES. For more information, see [“BULKLOAD= Table Option” on page 446](#).

The BULKOPTS= option is a container option that is required in order to specify other bulk load table options.

See “FedSQL Statement Table Options by Data Source” on page 418 for a list of the options that can be specified in this container for each data source.

Examples

Example 1

The following example uses the BULKLOAD=YES and BULKOPTS= options to submit bulk loading options to DB2 under UNIX and PC:

```
BULKLOAD=YES BULKOPTS=(BL_LOG='c:\temp\bulkload.log' BL_LOAD_REPLACE=yes
    BL_OPTIONS='ERRORS=999, LOAD=2000');
```

Example 2

The following example uses the BULKLOAD=YES and BULKOPTS= options to submit bulk loading options to Teradata. The Teradata Parallel Transporter (TPT) is used to load data into Teradata tables when BULKLOAD=YES. The TPT job uses the Stream operator by default. In the example below, the TD_TPT_OPER= table option specifies to use the Load operator instead, among other things.

```
BULKLOAD=YES BULKOPTS=(TD_TPT_OPER=LOAD TD_ERROR_LIMIT=1
    TD_TRACE_OUTPUT=tptrace TD_TRACE_LEVEL=TD_OPER_ALL
    TD_MAX_SESSIONS=8 TD_TRACE_LEVEL_INF=TD_OPER_ALL)
```

Example 3

The following example uses the BULKLOAD=YES and BL_BULKOPTS= options to submit bulk loading options to Oracle:

```
BULKLOAD=YES BULKOPTS=(BL_DEFAULT_DIR='C:\mylogs' BL_LOGFILE='novemberdatalog'
    BL_ERRORS=999)
```

See Also

Table Options:

- “TD_TPT_OPER= Table Option” on page 493

COMPRESS= Table Option

Specifies how rows are compressed in a new output data set.

Category: Table Control

Restrictions: Use with output tables only.
In SPD Engine data sets, cannot be used with ENCRYPT=YES or ENCRYPT=AES.

Data source: SAS data set, SPD Engine data set

Syntax

COMPRESS= NO | YES | CHAR | BINARY

Arguments

NO

specifies that the rows in a newly created table are uncompressed (fixed-length records).

YES | CHAR

specifies that the rows in a newly created table are compressed (variable length records). SAS data sets are compressed, by using Run Length Encoding (RLE). RLE compresses rows by reducing repeated consecutive characters (including blanks) to two-byte or three-byte representations. SPD Engine data sets are compressed by using the run length compression algorithm SPDSRLC2.

Alias ON

Tip Use this compression algorithm for character data.

BINARY

specifies that the rows in a newly created SAS data set are compressed by SAS by using Ross Data Compression (RDC). RDC combines run-length encoding and sliding-window compression to compress the table. SPD Engine data sets are compressed by the SPD Engine by using SPDSRDC.

Tip This method is highly effective for compressing medium to large (several hundred bytes or larger) blocks of binary data (numeric variables). Because the compression function operates on a single record at a time, the record length needs to be several hundred bytes or larger for effective compression.

Details

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

Use the COMPRESS= table option to compress an individual table. Specify the option for an output table only—that is, a table name on the CREATE TABLE statement.

Note: In SPD Engine data sets, encryption and compression are mutually exclusive.

You cannot specify encryption for a compressed SPD Engine data set. You cannot compress an encrypted SPD Engine data set.

After a table is compressed, the setting is a permanent attribute of the table, which means that to change the setting, you must re-create the table. That is, to uncompress a table, you must drop the table by using the DROP TABLE statement and re-create the table with the CREATE TABLE statement and the COMPRESS=NO option.

When you specify COMPRESS= to compress an SPD Engine data set, the table drivers compress, by blocks, the table as it is created. To specify the size of the compressed blocks, use the “[IOBLOCKSIZE= Table Option](#)” on page 463. The SPD Engine table driver also supports the PADCOMPRESS= table option when creating or updating the SPD Engine data set. The PADCOMPRESS= option enables you to add padding to the newly compressed blocks. See the

Comparisons

The COMPRESS= table option overrides the COMPRESS= connection string option.

SAS data sets only: When you create a compressed table, you can also specify the REUSE=YES table option in order to track and reuse space. With REUSE=YES, new rows are inserted in space freed when other rows are updated or deleted. When the default REUSE=NO is in effect, new rows are appended to the existing table.

See Also

Table Options:

- [“ENCRYPT= Table Option ” on page 453](#)
- [“POINTOBS= Table Option” on page 473](#)
- [“IOBLOCKSIZE= Table Option” on page 463](#)
- [“PADCOMPRESS= Table Option” on page 468](#)
- [“REUSE= Table Option” on page 475](#)

COPIES= Table Option

Specifies how many copies are made when file blocks are written to HDFS.

Category: Table Control

Data source: SASHDAT file

Syntax

COPIES=*number*

Arguments

number

specifies the number of copies for each file block (besides the original block) that you want stored in HDFS. The default value is one copy when data are loaded in parallel, and two copies if the NODIST(INNAMEONLY)= connection option is specified. Specifying 0 indicates that you do not want any replicates in HDFS.

DBCREATE_INDEX_OPTS=

Specifies DBMS-specific syntax to be added to the CREATE INDEX statement.

Category: Index Control

Data source: ODBC

Syntax

DBCREATE_INDEX_OPTS= *'DBMS-SQL-clauses'*

Arguments

DBMS-SQL-clauses

specifies one or more DBMS-specific clauses that can be appended to the end of an SQL CREATE INDEX statement.

Details

This option enables you to add DBMS-specific clauses to the end of the CREATE INDEX statement. The interface passes the CREATE INDEX statement and its clauses to the DBMS, which executes the statement and creates the DBMS index.

Example

In the following example, a Hive index is created with the value of the DBCREATE_INDEX_OPTS= “as 'compact' with deferred rebuild” option appended to the CREATE INDEX statement:

```
create index "COL1_1A03" on "TKTS002_1A03" {options DBCREATE_INDEX_OPTS="as 'compact'
with deferred rebuild"} ("COL1")
```

The following CREATE INDEX statement is passed to the DBMS in order to create the index:

```
create index 'COL1_1A03' on table 'TKTS002_1A03' ('COL1') as 'compact' with
deferred rebuild
```

DBCREATE_TABLE_OPTS= Table Option

Specifies DBMS-specific syntax to be added to the CREATE TABLE statement.

Category: Table Control

Data source: DB2 under UNIX and PC, Greenplum, MySQL, Netezza, Oracle, SAP HANA, Teradata

Syntax

DBCREATE_TABLE_OPTS= *'DBMS-SQL-clauses'*

Arguments

DBMS-SQL-clauses

specifies one or more DBMS-specific clauses that can be appended to the end of an SQL CREATE TABLE statement.

Details

This option enables you to add DBMS-specific clauses to the end of the CREATE TABLE statement. The interface passes the CREATE TABLE statement and its clauses to the DBMS, which executes the statement and creates the DBMS table.

The availability and behavior of FedSQL statement options are data-source specific. If DBCREATE_TABLE_OPTS= options are used in a statement other than CREATE TABLE, those options might be ignored.

Examples

Example 1

In the following example, the Greenplum table TEMP is created with the value of the DBCREATE_TABLE_OPTS='DISTRIBUTED BY ("B")' option appended to the CREATE TABLE statement:

```
create table temp {options dbcreate_table_opts='distributed by ("b")'}
(a int, b int);
```

The following CREATE TABLE statement is passed to the DBMS in order to create the table:

```
create table temp (a int, b int) distributed by ("b")
```

Example 2

In the following example, the MySQL table TEMP is created with the value of the DBCREATE_TABLE_OPTS='PARTITIONING KEY (x) USING HASHING' option appended to the CREATE TABLE statement:

```
create table temp {option dbcreate_table_opts='partitioning key (x) using hashing'}
(x double);
```

The following CREATE TABLE statement is passed to the DBMS in order to create the table:

```
create table temp (x double) partitioning key (x) using hashing;
```

Example 3

In the following example, the Netezza table TEMP is created with the value of the DBCREATE_TABLE_OPTS='DISTRIBUTE ON (b)' option appended to the CREATE TABLE statement:

```
create table temp {options dbcreate_table_opts='distribute on (b)'}
(a int, b int);
```

The following CREATE TABLE statement is passed to the DBMS in order to create the table:

```
create table temp (A int, B int) distribute on (b)
```

Example 4

In the following example, the Oracle table TEMP is created with the value of the DBCREATE_TABLE_OPTS='NOLOGGING' option appended to the CREATE TABLE statement:

```
create table temp{options dbcreate_table_opts='nologging'}
(a int, b int)
```

The following CREATE TABLE statement is passed to the DBMS in order to create the table:

```
create table temp ("a" number (10) , "b" number (10) ) nologging
```

Example 5

In the following example, the SAP HANA table TEMP is created with the value of the DBCREATE_TABLE_OPTS='PARTITION BY RANGE' option appended to the CREATE TABLE statement:

```
create table temp {options table_type=column dbcreate_table_opts="partition by range (x)
```

```
(partition 1 <= values < 10, partition others)"}
```

```
(x int, y int);
```

The following CREATE TABLE statement is passed to the DBMS in order to create the table:

```
create column table temp (x int, y int) partition by range (x)
(partition 1 <= values < 10, partition others)
```

Example 6

In the following example, the Teradata table TEMP is created with the value of the DBCREATE_TABLE_OPTS='PRIMARY INDEX (B)' option appended to the CREATE TABLE statement:

```
create table temp {options dbcreate_table_opts='primary index(b)'}
(a int, b int);
```

The following CREATE TABLE statement is passed to the DBMS in order to create the table:

```
create table temp (a int, b int) primary index(b)
```

See Also

Table Options:

- [“TABLE_TYPE=” on page 477](#)

ENCRYPT= Table Option

Specifies whether to encrypt an output SAS data set or SPD Engine data set.

Category: Table Control

Restrictions: Use with output tables only.
In SPD Engine data sets, cannot be used with COMPRESS=YES | CHAR | BINARY.

Data source: SAS data set, SPD Engine data set

Syntax

ENCRYPT= [AES](#) | [NO](#) | [YES](#)

Arguments

AES

encrypts the file using AES (Advanced Encryption Standard) algorithm. AES provides enhanced encryption by using SAS/SECURE software. SAS/SECURE is a product within the SAS System. Beginning with SAS 9.4, SAS/SECURE is included with Base SAS software. Before SAS 9.4, SAS/SECURE was an add-on product that was licensed separately. This change makes strong encryption available in all deployments (except where prohibited by import restrictions). You must specify the ENCRYPTKEY= table option when using ENCRYPT=AES. For more information, see [“ENCRYPTKEY= Table Option” on page 455](#).

Requirement SAS/SECURE software must be in use.

CAUTION

Record all ENCRYPTKEY= values when using ENCRYPT=AES. If you forget the ENCRYPTKEY= value, you lose your data. SAS cannot assist you in recovering the ENCRYPTKEY= value. The following note is written to the log:

Note: If you lose or forget the ENCRYPTKEY= value, there will be no way to open the file or recover the data.

NO

does not encrypt the data set.

YES

encrypts the data set using the SASProprietary algorithm. This encryption method uses passwords that are stored in the data set. At a minimum, you must specify the READ= or the PW= table option at the same time that you specify ENCRYPT=YES. Because the encryption method uses passwords, you cannot change *any* password on an encrypted file without re-creating the table.

CAUTION:

Record all passwords when using ENCRYPT=YES. If you forget the passwords, you cannot reset them.

Details

In SPD Engine data sets, encryption and compression are mutually exclusive. You cannot specify encryption for a compressed SPD Engine data set. You cannot compress an encrypted SPD Engine data set.

When using ENCRYPT=YES, the following rules apply:

- You can use the ENCRYPT= option only when you are creating a table.
- In order to copy an encrypted data file, make sure that the output engine supports encryption. Otherwise, the data file is not copied.
- Encrypted files work only in SAS 6.11 or later.
- If the data file is encrypted, all associated indexes are also encrypted.
- Encryption requires approximately the same amount of CPU resources as compression.
- You cannot use PROC CPORT on SASProprietary encrypted data files.

When using ENCRYPT=AES, the following rules apply:

- You must use the ENCRYPTKEY= table option when creating a table.
- To copy an encrypted AES data file, the output engine must support AES encryption. Otherwise, the data file is not copied.
- Beginning with SAS 9.4, you can use an encrypted AES data file.
- You must have SAS/SECURE software to use AES encryption.
- You cannot change the ENCRYPTKEY= value on an AES encrypted data file without re-creating the data file.

Example: Using the ENCRYPT=YES Option

This example creates an encrypted data set using the SASProprietary algorithm:

```
create table myfiles.salary {option encrypt=yes read=green}
    (name char(15),
```

```

        yrsal double,
        bonuspct double);
insert into myfiles.salary {option read=green} values ('Muriel', 34567, 3.2);
insert into myfiles.salary {option read=green} values ('Bjorn', 74644, 2.5);
insert into myfiles.salary {option read=green} values ('Freda', 38755, 4.1);
insert into myfiles.salary {option read=green} values ('Benny', 29855, 3.5);
insert into myfiles.salary {option read=green} values ('Agnetha', 70998, 4.1);

```

To retrieve data from the data set, you must specify the Read password:

```
select * from myfiles.salary {option read=green};
```

See Also

Table Options:

- [“ENCRYPTKEY= Table Option” on page 455](#)

ENCRYPTKEY= Table Option

Specifies a key value for AES encryption.

Category:	Table Control
Restriction:	ENCRYPT=AES must be set.
Requirement:	This option is required when ENCRYPT=AES is set.
Interaction:	You cannot change the key value on an AES encrypted data set without re-creating the data set.
Data source:	SAS data set, SPD Engine data set
Note:	Check your log after this operation to ensure that the encrypt key values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in <i>SAS Language Reference: Concepts</i> .

Syntax

ENCRYPTKEY= *key-value*

Syntax Description

key-value

assigns an encrypt key value. The key value can be up to 64 bytes long. You are able to create an ENCRYPTKEY= key value with or without quotation marks using the following rules:

no quotation marks

- alphanumeric characters and underscores only
- up to 64- bytes
- uppercase and lowercase letters
- must start with a letter
- no blank spaces
- is not case-sensitive

Example `encryptkey=key-value`
`encryptkey=key-value1`

single quotation marks

- alphanumeric, special, and DBCS characters
- up to 64 bytes
- uppercase and lowercase letters
- blank spaces, but not all blank
- is case-sensitive

Example `encryptkey='key-value'`
`encryptkey='1234*##mykey'`

double quotation marks

- alphanumeric, special, and DBCS characters
- up to 64 bytes
- uppercase and lowercase letters
- enables macro resolution
- blank spaces, but not all blanks
- is case-sensitive

Example `encryptkey="key-value"`
`encryptkey="1234*##mykey"`

```
%let mykey=abcdefghi12;
encryptkey=&key-value
```

Note When the ENCRYPTKEY= key value uses DBCS characters, the 64-byte limit applies to the character string after it has been transcoded to UTF-8 encoding. You can use the following DATA step to calculate the length in bytes of a key value in DBCS:

```
data _null_;
    key=length(unicodec("key-value", "UTF8"));
    put "key length=" key;
run;
```

Details

CAUTION:

Record the key value. If you forget the ENCRYPTKEY= key value, you lose your data. SAS cannot assist you in recovering the ENCRYPTKEY= key value because the key value is not stored with the data set. The following note is written to the log:

Note: If you lose or forget the ENCRYPTKEY= value,
there will be no way to open the file or recover the data.

You must use the ENCRYPTKEY= option when you are creating or accessing a SAS data set with AES encryption.

The ENCRYPTKEY= table option does not protect the file from deletion or replacement. Encrypted data sets can be deleted by using any of the following scenarios without having to specify an ENCRYPTKEY= key value:

- the KILL option in PROC DATASETS
- the DROP statement in PROC SQL
- the DELETE procedure
- the DROP TABLE statement in FedSQL

The ENCRYPTKEY= option only prevents access to the contents of the file. To protect the file from deletion or replacement, the file must also contain an ALTER= password.

The following DATASETS procedure statements require you to specify the ENCRYPTKEY= key value when working with protected files: AGE, AUDIT, APPEND, CHANGE, CONTENTS, MODIFY, REBUILD, and REPAIR statements.

```
append base=name data=name(encryptkey=key-value);
run;
```

The option can be specified either in parentheses after the name of the SAS data file or after a forward slash.

It is possible to use a macro variable as the ENCRYPTKEY= key value. When you specify a macro variable for the ENCRYPTKEY= key value, you must enclose the macro variable in double quotation marks. If you do not use the double quotation marks, unpredictable results can occur. The following example defines a macro variable and uses the macro variable as the ENCRYPTKEY= key value:

```
%let secret=myvalue;
create table myschema.dsname {options encrypt=aes encryptkey="&secret"};
```

Example: Using the ENCRYPTKEY= Option

This example uses the ENCRYPT=AES option to encrypt a SAS data set:

```
create table myfiles.salary {options encrypt=aes encryptkey="1234*##mykey"}
  (name char(15),
   yrsal double,
   bonuspct double);
insert into myfiles.salary {option encryptkey="1234*##mykey"} values
  ('Muriel', 34567, 3.2);
insert into myfiles.salary {option encryptkey="1234*##mykey"} values
  ('Bjorn', 74644, 2.5);
insert into myfiles.salary {option encryptkey="1234*##mykey"} values
  ('Freda', 38755, 4.1);
insert into myfiles.salary {option encryptkey="1234*##mykey"} values
  ('Benny', 29855, 3.5);
insert into myfiles.salary {option encryptkey="1234*##mykey"} values
  ('Agnetha', 70998, 4.1);
```

To retrieve data from the data set, you must specify the Read password:

```
select * from myfiles.salary {option encryptkey="1234*##mykey"};
```

See Also

Table Options:

- [“ENCRYPT= Table Option ” on page 453](#)

ENDOBS= Table Option

Specifies the end observation number in a user-defined range of observations to be processed.

Used by:	STARTOBS= table option
Category:	Observation Control
Restriction:	Use with input data sets only
Data source:	SPD Engine data set

Syntax

ENDOBS= *n*

Arguments

n
is the number of the end observation.

Details

The software processes all of the observations in the entire data set unless you specify a range of observations with the STARTOBS= or ENDOBS= options. If the STARTOBS= option is used without the ENDOBS= option, the implied value of ENDOBS= is the end of the data set. When both options are used together, the value of ENDOBS= must be greater than the value of STARTOBS=.

When ENDOBS= is used in a WHERE expression, the ENDOBS= value represents the last observation to process, rather than the number of observations to return.

See Also

Table Options:

- [“STARTOBS= Table Option” on page 476](#)

EXTENDOBSCOUNTER= Table Option

Specifies whether to extend the maximum observation count in a new output SAS data set.

Valid in:	CREATE TABLE statement
Category:	Table Control
Alias:	EOC=
Default:	YES
Data source:	SAS data set

Syntax

EXTENDOBSCOUNTER=YES | NO

Arguments

YES

requests an enhanced file format in a newly created SAS data set that counts observations beyond the 32-bit limitation. Although this SAS data set is created for an operating environment that stores the number of observations with a 32-bit integer, the data set behaves like a 64-bit file with respect to counters. This is the default value.

Restrictions EXTENDOBSCOUNTER=YES is valid only for an output SAS data set whose internal data representation stores the observation count as a 32-bit integer. EXTENDOBSCOUNTER= is ignored for SAS data sets with a 64-bit integer.

A SAS data set that is created with an extended observation count is incompatible with releases prior to SAS 9.3.

NO

specifies that the maximum observation count in a newly created SAS data file is determined by the long integer size for the operating environment. In operating environments with a 32-bit integer, the maximum number is $2^{31}-1$ or approximately two billion observations (2,147,483,647). In operating environments with a 64-bit integer, the maximum number is $2^{63}-1$ or approximately 9.2 quintillion observations.

Details

Historically, Base SAS had a limitation in which up to approximately two billion observations could be counted and fully supported for operating environments with a 32-bit long integer. The EXTENDOBSCOUNTER= table option extends the limit to match that of operating environments with a 64-bit long integer. EXTENDOBSCOUNTER=NO is provided for backward compatibility.

GP_DISTRIBUTED_BY= Table Option

Specifies the distribution key for the table being created.

Category: Table Control

Alias: DISTRIBUTED_BY=

Data source: Greenplum

Syntax

GP_DISTRIBUTED_BY= 'DISTRIBUTED BY (*column*[, ...*column*])
| DISTRIBUTED RANDOMLY'

Arguments

DISTRIBUTED BY (*column*, <...*column*>)

specifies one or more DBMS column names to use as the distribution key.

DISTRIBUTED RANDOMLY

specifies to determine the column or set of columns to use to distribute table rows across database segments. This is known as round-robin distribution.

Details

DISTRIBUTED BY uses hash distribution with one or more columns declared as the distribution key. For the most even data distribution, the distribution key should be the primary key of the table or a unique column (or set of columns). If that is not possible, then you might choose DISTRIBUTED RANDOMLY, which sends the data round-robin to the segment instances. If a value is not supplied, then hash distribution is chosen using the primary key (if the table has one) or the first eligible column of the table as the distribution key.

DISTRIBUTED_BY can be submitted as shown above or within the DBCREATE_TABLE_OPTS= table option. Here is an example of how it is specified in DBCREATE_TABLE_OPTS=:

```
dbcreate_table_opts='distributed by ("b")'
```

See Also

Table Options:

- [“DBCREATE_TABLE_OPTS= Table Option” on page 451](#)

HASH= Table Option

Specifies that when partitioning data, the distribution of partitions is not determined by a tree, but by a hashing algorithm.

Category: Table Control

Default: NO

Interaction: Valid only with the PARTITION= table option.

Data source: SASHDAT file

Syntax

```
PARTITION=(column-list) HASH=[YES | NO]
```

Arguments

YES

Invokes a hash function to determine the distribution properties of the partitions.

NO

Specifies that the distribution scheme depends on a binary tree.

Details

Specifying HASH=YES is recommended when you work with high-cardinality partition keys (in the order of millions of partitions). When HASH=YES is used, the partitions are not as balanced, but result in lower memory usage.

Example

```
create table sashdat.transactions{OPTIONS partition=(cust_id, year) hash=yes}
```

```
as select * from someschema.sometable;
```

See Also

Table Options:

- [“PARTITION= Table Option” on page 471](#)

IDXNAME= Table Option

Directs SAS to use a specific index to match the conditions of a WHERE clause.

Category: User Control of SAS Index Usage

Restrictions: Use with input data sets only
Cannot be used with the IDXWHERE= table option

Data source: SAS data set

Syntax

IDXNAME= *index-name*

Arguments

index-name

specifies the name (up to 32 characters) of a simple or composite index for the SAS data set. SAS does not attempt to determine whether the specified index is the best one or whether a sequential search might be more resource efficient.

Interaction The specification is not a permanent attribute of the data set and is valid only for the current use of the data set.

Details

By default, to satisfy the conditions of a WHERE clause for an indexed SAS data set, SAS identifies zero or more candidate indexes that could be used to optimize the WHERE clause. From the list of candidate indexes, SAS selects the one that it determines will provide the best performance, or rejects all of the indexes if a sequential pass of the data is expected to be more efficient.

Because the index that SAS selects might not always provide the best optimization, you can direct SAS to use one of the candidate indexes by specifying the IDXNAME= table option. If you specify an index that SAS does not identify as a candidate index, then IDXNAME= table option does not process the request. That is, IDXNAME= does not allow you to specify an index that would produce incorrect results.

Comparisons

The IDXWHERE= table option enables you to override the SAS decision about whether to use an index.

Example

This example uses the `IDXNAME=` table option to direct SAS to use a specific index to optimize the `WHERE` clause. SAS then disregards the possibility that a sequential search of the data set might be more resource efficient and does not attempt to determine whether the specified index is the best one. (Note that the `EMPNUM` index was not created with the `NOMISS` option.)

```
create table mydata.empnew
  as select * from mydata.employee {option idxname=empnum}
  where empnum < 2000;
```

See Also

Table options

- [“IDXWHERE= Table Option” on page 462](#)

IDXWHERE= Table Option

Specifies whether SAS uses an index search or a sequential search to match the conditions of a `WHERE` clause.

Category:	User Control of SAS Index Usage
Restrictions:	Use with input data sets only SAS data sets: Cannot be used with <code>IDXNAME=</code> SPD Engine data sets: <code>IDXWHERE=NO</code> cannot be used with <code>WHEREINDEX=</code>
Data source:	SAS data set, SPD Engine data set

Syntax

IDXWHERE= [YES](#) | [NO](#)

Arguments

YES

tells SAS to choose the best index to optimize a `WHERE` clause, and to disregard the possibility that a sequential search of the data set might be more resource-efficient. This is the default value.

NO

tells SAS to ignore all indexes and satisfy the conditions of a `WHERE` clause with a sequential search of the data set.

Notes You cannot use the `IDXWHERE=` table option to override the use of an index to process a `BY` statement.

You cannot use the `WHEREINDEX=` table option when `IDXWHERE=NO` is used.

Details

By default, to satisfy the conditions of a WHERE clause for an indexed data set, the software decides whether to use an index or to read the data set sequentially. The software estimates the relative efficiency and chooses the method that is more efficient.

You might need to override the software's decision by specifying the IDXWHERE= table option because the decision is based on general rules that occasionally might not produce the best results. That is, by specifying the IDXWHERE= table option, you are able to determine the processing method.

Note: The specification is not a permanent attribute of the data set and is valid only for the current use of the data set.

Comparisons

The IDXNAME= table option (which is supported only for SAS data sets) enables you to direct SAS to use a specific index.

The WHERENOINDEX= table option enables you to specify a list of indexes to exclude when making WHERE expression evaluations.

Examples

Example 1: Specifying Index Usage

This example uses the IDXWHERE= table option to tell SAS to decide which index is the best to optimize the WHERE clause. SAS then disregards the possibility that a sequential search of the data set might be more resource-efficient:

```
create table mydata.empnew
  as select * from mydata.employee {option idxwhere=yes}
  where empnum < 2000;
```

Example 2: Specifying No Index Usage

This example uses the IDXWHERE= table option to tell SAS to ignore any index and to satisfy the conditions of the WHERE clause with a sequential search of the data set:

```
create table mydata.empnew
  as select * from mydata.employee {option idxwhere=no}
  where empnum < 2000;
```

See Also

Table options:

- [“IDXNAME= Table Option” on page 461](#)
- [“WHERENOINDEX= Table Option” on page 499](#)

IOBLOCKSIZE= Table Option

Specifies the number of rows in a block to be used in an I/O operation.

Category: Table Control

Data source: SPD Engine data set

Syntax

IOBLOCKSIZE= *n*

Arguments

n

specifies the size of the block in bytes. The default value is 32,768 bytes.

Details

The software reads and stores rows in the table in blocks. IOBLOCKSIZE= is useful on compressed or encrypted tables. The software does not use IOBLOCKSIZE= on noncompressed or nonencrypted tables.

For tables that you compress or encrypt, the IOBLOCKSIZE= specification determines the number of rows to include in the block. The specification applies to block compression as well as data I/O to and from disk. The IOBLOCKSIZE= value affects the table's organization on disk.

When using compression or encryption, specify an IOBLOCKSIZE= value that complements how the data is to be accessed, sequentially or randomly. Sequential access or operations requiring full table scans favor a large block size. On SPD Engine, a large block size would be 131,072 bytes. In contrast, random access favors a smaller block size. The default value of 32,768 bytes is the smallest allowed value.

See Also

Table Options:

- “COMPRESS= Table Option ” on page 448
- “ENCRYPT= Table Option ” on page 453
- “PADCOMPRESS= Table Option” on page 468

LABEL= Table Option

Specifies a label for a table.

Category: Table Control

Data source: SAS data set, SASHDAT file, SPD Engine data set

Syntax

LABEL= '*label*'

Arguments

'label'

specifies a text string of up to 256 characters.

Requirements For SAS data sets and SPD Engine data sets, if the text contains single quotation marks, use double quotation marks around the

label. Or, use two single quotation marks in the label text and enclose the string in single quotation marks. To remove a label from a data set, assign a label that is equal to a blank that is enclosed in quotation marks.

Always use single quotation marks for SASHDAT files.

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. The label can be viewed with the SAS CONTENTS procedure.

You can use the LABEL= table option on both input and output tables. When you use the LABEL= table option on input tables, it assigns a label for the file for the duration of the operation. When this is specified for an output table, the label becomes a permanent part of that file.

A label assigned to a table remains associated with that table when you update a table in place, such as when you use the APPEND procedure or the MODIFY statement. However, a label is lost if you use a table with a previously assigned label to create a new table with the CREATE TABLE statement. For example, a label previously assigned to table ONE is lost when you create the new output table ONE in this CREATE TABLE statement:

```
create table one as select * from one;
```

Example

These examples assign labels to SAS data sets:

```
create table tst {option label='1976 W2 Info, Hourly'} (col1 char(4), col2 double);
create table tst1 {option label="Hillside's Daily Account"} as select from tst3;
create table tst2 {option label='Peter''s List'}(col1 char(20), col2 char(20));
```

LOCKTABLE= Table Option

Places shared or exclusive locks on tables.

Category: Table Control

Data source: SAS data set

Syntax

LOCKTABLE= [SHARE](#) | [EXCLUSIVE](#)

Arguments

SHARE

locks a table in shared mode, allowing other users or processes to read data from the tables, but preventing users from updating data.

EXCLUSIVE

locks a table exclusively, preventing other users from accessing any table that you open.

Details

You can lock tables only if you are the owner or have been granted the necessary privilege.

If you access the BASE table driver through PROC FEDSQL, the default value for the LOCKTABLE option is EXCLUSIVE. However, if you access the BASE table driver through a SAS Federation Server, or if you run your program locally with the SAS Federation Server LIBNAME engine, the default value for the LOCKTABLE option is SHARE.

ORDERBY= Table Option

Specifies the columns by which to order the data within a partition.

Category: Table Control

Interaction: Valid only with the PARTITION= table option.

Data source: SASHDAT file

Syntax

ORDERBY=(*column-list*)

ORDERBY=(*column* [DESCENDING])

Arguments***column-list***

specifies the columns by which to order the data within a partition. The specified column(s) must exist and cannot be partitioning columns. When multiple columns are specified, the column names must be separated by commas.

Details

The ordering is hierarchical. For example, ORDERBY=(A, B) specifies ordering by the values of column B within the ordered values of column A. The order is determined based on the raw value of the columns and uses locale-sensitive collation for character columns.

By default, column values are arranged in ascending order. You achieve descending order by specifying the keyword DESCENDING or DESC after the column name in the column list.

Example

```
create table sashdat.employees {options partition=(dept)
orderby=(hire_date descending)} as select * from mybase.employees;
```

See Also

Table Options:

- [“PARTITION= Table Option” on page 471](#)
- [“UCA= Table Option” on page 498](#)

ORHINTS= Table Option

Specifies Oracle hints to pass to Oracle from FedSQL.

Valid in:	DELETE statement, INSERT statement, SELECT statement, UPDATE statement
Category:	Data control
Data source:	Oracle

Syntax

ORHINTS=*/* Oracle-hint */*

Arguments

Oracle-hint

specifies an Oracle hint for the FedSQL language processor to pass to the DBMS as part of a query. If you do not specify a hint, FedSQL does not send any hints to Oracle. For more information, see *SAS Federation Server: Administrator's Guide*.

Details

The ORHINTS= table option is used in conjunction with the DRIVER_TRACE= and DRIVER_TRACEFILE= data source connection options. PROC FEDSQL and PROC DS2 do not support use of data source connection options.

Example: Using the ORHINTS= Option

These examples show how to specify ORHINTS=:

```
select * from test{options orhints='/* dummy hint */'} ;

update test{options orhints='/* dummy hint */'} set c1=1;

insert into test{options orhints='/* dummy hint */'} values (100);

delete from test{options orhints='/* dummy hint */'} ;
```

ORNUMERIC= Table Option

Specifies how numbers read from or inserted into the Oracle NUMBER column are treated.

Category:	Table Control
Default:	YES

Data source: Oracle
Data type: DECIMAL, NUMERIC

Syntax

ORNUMERIC=YES | NO

Arguments

NO

Indicates that the numbers are treated as TKTS_DOUBLE values. They might not have precision beyond 14 digits.

YES

Indicates that non-integer values with explicit precision are treated as TKTS_NUMERIC values. This is the default setting.

Details

This option defaults to YES so that a NUMBER column with precision or scale is described as TKTS_NUMERIC. This option can be specified as both a connection option and a table option. When specified as both connection and table option, the table option value overrides the connection option.

PADCOMPRESS= Table Option

Specifies the number of bytes to add to compressed blocks in an SPD Engine data set that is opened for OUTPUT or UPDATE.

Category: Table Control
Data source: SPD Engine data set

Syntax

PADCOMPRESS= *n*

Arguments

n

specifies the number of bytes to add. The default number is 0 (zero).

Details

Compressed SPD Engine data sets occupy blocks of space on the disk. The size of a block is derived from the IOBLOCKSIZE= table option that is specified when the data set is created. When the data set is updated, a new block fragment might need to be created to hold the update. More updates might then create new fragments, which, in turn, increases the number of I/O operations needed to read a data set.

By increasing the block padding in certain situations where many updates to the data set are expected, fragmentation can be kept to a minimum. However, adding padding can waste space if you do not update the data set.

You must weigh the cost of padding all compression blocks against the cost of possible fragmentation of some compression blocks.

Specifying the PADCOMPRESS= table option when you create or update a data set adds space to all of the blocks as they are written back to the disk. The PADCOMPRESS= setting is not retained in the data set's metadata.

See Also

Table Options:

- “COMPRESS= Table Option ” on page 448
- “IOBLOCKSIZE= Table Option” on page 463

PARTSIZE= Table Option

Specifies the size of the table partitions.

Valid in: CREATE TABLE statement

Category: Table Control

Data source: SPD Engine data set

Syntax

PARTSIZE= *n*

Arguments

n

specifies the size of the partition.

Requirements The size can be specified in megabytes, gigabytes, and terabytes. If *n* is specified without M, G, or T, the default is megabytes. For example, PARTSIZE=128 is the same as PARTSIZE=128M. The default value is 128 megabytes. The maximum value is 8,796,093,022,207 megabytes.

When creating an SPD Engine data set, if the row length is greater than 65K, you might find that the PARTSIZE= that you specify and the actual partition size do not match. To get these numbers to match, specify a PARTSIZE= that is a multiple of 32 and the row length.

Details

The option PARTSIZE= forces the software to partition data files at the specified size. The actual size of the partition is computed to accommodate the maximum number of rows that fit in the specified size of *n*.

Multiple partitions are necessary to read the data in SPD Engine data sets in parallel. The value is specified when an SPD Engine data set is created. This size is a fixed size. This specification applies only to the data component files. By splitting (partitioning) the data portion of a data set into fixed-sized files, the engine can introduce a high degree of

scalability for some operations. The engine can spawn threads in parallel (for example, up to one thread per partition for WHERE evaluations). Separate data partitions also enable the engine to process the data without the overhead of file access contention between the threads. Because each partition is one file, the trade-off for a small partition size is that an increased number of files (for example, UNIX i-nodes) are required to store the observations.

The partition size determines a unit of work for many of the parallel operations that require full data set scans. But, having more partitions does not always mean faster processing. The trade-offs involve balancing the increased number of physical files (partitions) that are required to store the data set against the amount of work that can be done in parallel by having more partitions. More partitions means more open files to process the data set, but a smaller number of rows in each partition.

A general rule is to have 10 or fewer partitions per data path, and 3 to 4 partitions per CPU. (Some operating systems have a limit on the number of open files that you can use.)

To determine an adequate partition size for a new SPD Engine data set, you should be aware of the following:

- the types of applications that run against the data
- how much data you have
- how many CPUs are available to the applications
- which disks are available for storing the partitions
- the relationships of these disks to the CPUs

For example, if each CPU controls only one disk, then an appropriate partition size would be one in which each disk contains approximately the same amount of data. If each CPU controls two disks, then an appropriate partition size would be one in which the load is balanced. Each CPU does approximately the same amount of work.

Ultimately, scalability limits using PARTSIZE= depend on how you structure the DATAPATH=. See information about the DATAPATH= LIBNAME option in *SAS Scalable Performance Data Engine: Reference*. Specifically, the limits depend on how you configure and spread the DATAPATH= file systems across striped volumes. You should spread each individual volume's striping configuration across multiple disk controllers or SCSI channels in the disk storage array. The goal for the configuration is, at the hardware level, to maximize parallelism during data retrieval.

The PARTSIZE= specification is limited by the MINPARTSIZE= system option. MINPARTSIZE= ensures that an over-zealous SAS user does not create arbitrarily small partitions, thereby generating a large number of files. The default MINPARTSIZE= for SPD Engine data sets is 16 megabytes and probably should not be lowered much beyond this value.

Note: The PARTSIZE= value for a data set cannot be changed after a data set is created. To change the PARTSIZE=, you must delete the table with the DROP TABLE statement and create it again with the CREATE TABLE statement.

Example: Specifying PARTSIZE= for an SPD Engine Data Set

You have 100 gigabytes of data and 8 disks, so you can store 12.5 gigabytes per disk. Optimally, you want 3 to 4 partitions per disk. A partition size of 3.125 gigabytes is appropriate. So, you can specify PARTSIZE=3200M.

```
create table salecent.sw {options partsize=3200m};
```

PARTITION= Table Option

Specifies the columns to use to partition the SASHDAT file.

Category: Table Control

Data source: SASHDAT file

Syntax

PARTITION=(*column-list*)

Arguments

column-list

specifies the column or columns to use to partition the SASHDAT file. The columns must already exist and cannot have been specified in the ORDERBY= table option. When multiple columns are specified, the column names must be separated by commas.

Details

Partitioning is available only when you create tables.

Partition keys are derived based on formatted values in the order of the column names in the *column-list*. All of the rows with the same partition key are stored in a single block. This ensures that all the data for a partition is loaded into memory on a single machine in the cluster. The blocks are replicated according to the default replication factor or the value that you specify in the COPIES= table option.

The partitioning columns, whether they are numeric or character, are formatted by applying SAS formatting rules. For more information, see [Chapter 4, “FedSQL Formats,” on page 67](#). If user-defined formats are used, then the format name is stored with the table, but not the format. The format for the column must be available to the SAS LASR Analytic Server when the table is loaded into memory.

The key construction is not hierarchical. That is, PARTITION=(A, B) implies that any unique concatenation of formatted values for columns A and B defines a partition.

There is no ordering of the rows within a partition. You can create ordering within a partition by specifying the ORDERBY= option.

Example

The following example uses the PARTITION= table option to partition a table by the character variable REGION and the numeric variable DATE, where the date is formatted with the MONNAME3. format. The formatted values of the columns appear in the order in which the columns are specified in the PARTITION= option. Therefore, the keys for this example might look something like this: “EastJan”, “WestJan”, “NorthJan”, “SouthJan”, “EastFeb”, “WestFeb”, and so on.

```
create table sales(options partition=(region,"date"))
(region char(10), "date" date having format monname3.);
```

See Also

Table Options:

- [“HASH= Table Option” on page 460](#)
- [“ORDERBY= Table Option” on page 466](#)

PARTITION_KEY= Table Option

Specifies the column name to use as the partition key for creating fact tables.

Category: Table Control

Data source: Aster

Syntax

PARTITION_KEY= "*column-name*"

Arguments

column-name

specifies the name of the column, in quotation marks.

Details

Aster uses two table types: dimension and fact. To create a fact table in Aster without error, you must set the PARTITION_KEY= table option.

PERM= Table Option

Specifies the permission setting when writing a SASHDAT file to HDFS.

Category: Security

Alias: PERMISSION=

Data source: SASHDAT file

Syntax

PERM=*mode*

Arguments

mode

The mode is specified as an integer (for example, PERM=755). The integer is converted to a proper umask by the software. If no permission is specified, the access permissions for the table are set according to the umask of the user that loads the table.

POINTOBS= Table Option

Specifies whether SAS creates compressed data sets whose observations can be randomly accessed or sequentially accessed.

Category:	Table Control
Restriction:	Use with output tables only.
Interaction:	Used with COMPRESS= table option
Data source:	SAS data set

Syntax

POINTOBS= YES | NO

Syntax Description

YES

causes the FedSQL language to produce a compressed data set that might be randomly accessed by observation number. This is the default.

Here are examples of accessing data directly by observation number:

- the POINT= option of the MODIFY and SET statements in the DATA step
- going directly to a specific observation number with PROC FSEDIT.

TIP Specifying POINTOBS=YES does not affect the efficiency of retrieving information from a data set. It does increase CPU usage by approximately 10% when creating a compressed data set and when updating or adding information to it.

NO

suppresses the ability to randomly access observations in a compressed data set by observation number.

TIP Specifying POINTOBS=NO is desirable for applications where the ability to point directly to an observation by number within a compressed data set is not important. If you do not need to access data by observation number, then you can improve performance by approximately 10% by specifying POINTOBS=NO:

- when creating a compressed data set
- when updating or adding observations to it

Details

Note that REUSE=YES takes precedence over POINTOBS=YES. For example:

```
create table test{options compress=yes pointobs=yes reuse=yes};
```

This example code results in a data set that has POINTOBS=NO. Because POINTOBS=YES is the default when you use compression, REUSE=YES causes POINTOBS= to change to NO.

See Also

Table Options:

- “COMPRESS= Table Option ” on page 448
- “REUSE= Table Option” on page 475

PW= Table Option

Assigns a READ, WRITE, and ALTER password to a SAS data set or an SPD Engine data set and enables access to a password-protected file.

Category: Table Control

Data source: SAS data set, SPD Engine data set

Note: Check your log after this operation to ensure that the password values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in *SAS Language Reference: Concepts*.

Syntax

PW= *password*

Arguments

password

must be a valid SAS name.

Details

The PW= table option applies only to a SAS data set or an SPD Engine data set. You can use this option to assign a password or to access a password-protected file. When a data set that is protected by a password is replaced, the new data set inherits the password.

Note: A SAS password does not control access to a SAS file beyond the SAS system. You should use the operating system-supplied utilities and file-system security controls in order to control access to SAS files outside of SAS.

READ= Table Option

Assigns a READ password to a SAS data set or an SPD Engine data set that prevents users from reading the file, unless they enter the password.

Category: Table Control

Data source: SAS data set, SPD Engine data set

Note: Check your log after this operation to ensure that the password values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in *SAS Language Reference: Concepts*.

Syntax

READ= *read-password*

Arguments

read-password

must be a valid SAS name.

Details

The READ= option applies only to a SAS data set or an SPD Engine data set. You can use this option to assign a password or to access a Read-protected file.

Note: A SAS password does not control access to a SAS file beyond the SAS system. You should use the operating system-supplied utilities and file-system security controls in order to control access to SAS files outside of SAS.

REUSE= Table Option

Specifies whether new rows can be written to freed space in a compressed SAS data set.

Category: Table Control

Restriction: Use with output data sets only

Data source: SAS data set

Syntax

REUSE= NO | YES

Arguments

NO

does not track and reuse space in a compressed SAS data set. New rows are appended to the existing data set. Specifying NO results in less efficient data storage if you delete or update many rows in the data set.

YES

tracks and reuses space in a compressed SAS data set. New rows are inserted in the space that is freed when other rows are updated or deleted.

If you plan to use operations that add rows to the end of a compressed data set, use REUSE=NO. REUSE=YES causes new rows to be added wherever there is space in the file, not necessarily at the end of the file.

Details

By default, new rows are appended to an existing compressed data set. If you want to track and reuse free space by deleting or updating other rows, use the REUSE= table option when you create a compressed SAS data set.

The REUSE= table option has meaning only when you are creating a new data set with the COMPRESS=YES option. Using the REUSE= table option when you are accessing an existing SAS data set has no effect.

See Also

Table Options:

- [“COMPRESS= Table Option ” on page 448](#)
- [“POINTOBS= Table Option” on page 473](#)

SQUEEZE= Table Option

Specifies whether to write the SASHDAT file in compressed format.

Category:	File Control
Restriction:	Requires a grid server that uses the second version of SAS 9.4 or greater
Data source:	SASHDAT file

Syntax

SQUEEZE=[YES | NO]

Arguments

NO

Write the SASHDAT file in regular, non-compressed format. This is the default value if the SQUEEZE= table option is omitted.

YES

Writes the SASHDAT file in compressed format.

Details

Compressing data saves space but imposes a performance penalty on operations performed on a compressed file.

STARTOBS= Table Option

Specifies the starting row number in a user-defined range of rows to be processed.

Valid in:	SELECT statement
Category:	Observation Control
Restriction:	Use STARTOBS= with input data sets only
Data source:	SPD Engine data set

Syntax

STARTOBS= *n*

Arguments

n

is the number of the starting row.

Details

The software processes all of the rows in the entire data set unless you specify a range of rows with the STARTOBS= and ENDOBS= options. If the ENDOBS= option is used without the STARTOBS= option, the implied value of STARTOBS= is 1. When both options are used together, the value of STARTOBS= must be less than the value of ENDOBS=.

When STARTOBS= is used in a WHERE expression, the STARTOBS= value represents the first row on which to apply the WHERE expression.

See Also

Table Options:

- [“ENDOB= Table Option” on page 458](#)

TABLE_TYPE=

Specifies the type of table storage FedSQL will use when creating tables in SAP HANA.

Valid in: CREATE TABLE statement

Category: Data Access

Interactions: GLOBAL and GLOBAL TEMPORARY have the same behavior
LOCAL and LOCAL TEMPORARY have the same behavior

Data source: SAP HANA

Syntax

TABLE_TYPE= [COLUMN] [GLOBAL] [GLOBAL TEMPORARY] [LOCAL]
[LOCAL TEMPORARY] [ROW]

Arguments

COLUMN

creates a table using column-based storage in SAP HANA.

GLOBAL | GLOBAL TEMPORARY

creates a global, temporary table in SAP HANA. The tables are globally available; however, the data is visible only in the current session.

LOCAL | LOCAL TEMPORARY

creates a local, temporary table in SAP HANA. The table definition and data are visible only in the current session.

ROW

creates a table using row-based storage in SAP HANA.

Details

The SAP HANA TABLE_TYPE= option is available as a connection option and as a table option. If neither are specified, tables that are created in SAP HANA follow the SAP HANA default for row or column store.

See Also

Table Options:

- [“DBC_CREATE_TABLE_OPTS= Table Option” on page 451](#)

TD_BUFFER_MODE= Table Option

Specifies whether the LOAD method is used.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified within the [“BULK_OPTS= Table Option” on page 447](#)

Data source: Teradata

Syntax

TD_BUFFER_MODE= YES | NO

Arguments

YES

enables the bulk load feature. This option must be set to YES for the bulk load feature to work.

NO

disables the bulk load feature. This is the default value.

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)

TD_CHECKPOINT= Table Option

Specifies when the TPT operation issues a checkpoint or savepoint to the database.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified within the [“BULK_OPTS= Table Option” on page 447](#)

Data source: Teradata

Syntax

TD_CHECKPOINT= *number*

Arguments

number

specifies the number of rows after which the TPT operation issues a checkpoint or savepoint to the database. The default is 0, which means no checkpoint is taken. All rows are saved at the end of the job.

TD_DATA_ENCRYPTION= Table Option

Activates data encryption.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified within the “[BULKOPTS= Table Option](#)” on page 447

Data source: Teradata

Syntax

TD_DATA_ENCRYPTION= [ON](#) | [OFF](#)

Arguments

ON

encrypts all SQL requests, responses, and data.

OFF

no encryption occurs. This is the default setting.

See Also

Table Options:

- “[BULKLOAD= Table Option](#)” on page 446

TD_DROP_ERROR_TABLE= Table Option

Drops the log table at the end of the job, whether the job was completed successfully or not.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified within the “[BULKOPTS= Table Option](#)” on page 447

Data source: Teradata

Syntax

TD_DROP_ERROR_TABLE= [YES](#) | [NO](#)

Arguments

YES

specifies to drop the error tables at the end of the job, whether the job was completed successfully or not.

NO

keeps error tables that contain errors after the job is complete. Error tables that are empty after successful completion are deleted. This is the default setting.

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)

TD_DROP_LOG_TABLE= Table Option

Drops the log table at the end of the job, whether the job completed successfully or not.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified within the [“BULKOPTS= Table Option” on page 447](#)

Data source: Teradata

Syntax

TD_DROP_LOG_TABLE= [YES](#) | [NO](#)

Arguments

YES

specifies to drop the log table at the end of the job, whether the job is completed successfully or not.

NO

keeps log tables for jobs that were not completed successfully. This is the default setting.

TD_DROP_WORK_TABLE= Table Option

Drops the work table at the end of the job, whether the job was completed successfully or not.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified within the [“BULKOPTS= Table Option” on page 447](#)

Data source: Teradata

Syntax

TD_DROP_WORK_TABLE= YES | NO

Arguments

YES

specifies to drop the work table at the end of the job, whether the job was completed successfully or not.

NO

keeps work tables for jobs that were not completed successfully. This is the default setting.

TD_ERROR_LIMIT= Table Option

Specifies the maximum number of records that can be stored in an error table.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified within the “[BULKOPTS= Table Option](#)” on page 447

Data source: Teradata

Syntax

TD_ERROR_LIMIT= *number-of-records*

Arguments

number-of-records

specifies the maximum number of records that can be stored in an error table before the Load, Stream, or Update operator job is terminated. By default, the ErrorLimit value is unlimited. The number of records must be greater than zero.

Details

The ErrorLimit specification applies to each instance of an operator job. Specifying an invalid value terminates the job.

See Also

Table Options:

- “[BULKLOAD= Table Option](#)” on page 446

TD_ERROR_TABLE_1= Table Option

Specifies a name for the first error table.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified within the “[BULKLOAD= Table Option](#)” on page 446

Data source: Teradata

Syntax

TD_ERROR_TABLE_1= *name-of-table*

Arguments

name-of-table

specifies a name for the error table. The default name for ErrorTable1 is *tname_ET*, where *tname* is the name of the target table. Table names exceeding 27 characters are truncated to accommodate the three-character suffix. Therefore, you might want to specify a name for the table that will not be truncated. When truncation occurs, a message is written to the log.

Restriction The name of an existing table cannot be used, unless you are restarting a paused job.

Details

ErrorTable1 contains records that were rejected during the acquisition phase of a Load, Stream, or Update operator job because of the following:

- Data conversion errors
- Constraint violations
- Access Module Processor (AMP) configuration changes.

The error table is created in the default user (logon) database, optionally qualified with a schema, unless the TD_LOGDB= table option is used to specify a different location for utility tables.

Error tables that contain errors are retained at the end of a job. Specify the TD_DROP_ERROR_TABLE= table option if you do not want to retain them.

For information about the error table format and the procedure to correct errors, see *Teradata Parallel Transporter Reference*.

See Also

Table Options:

- “[BULKLOAD= Table Option](#)” on page 446
- “[TD_DROP_ERROR_TABLE= Table Option](#)” on page 479
- “[TD_LOGDB= Table Option](#)” on page 485

TD_ERROR_TABLE_2= Table Option

Specifies a name for the second error table.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified within the “BULKOPTS= Table Option” on page 447
Data source:	Teradata

Syntax

TD_ERROR_TABLE_2= *name-of-table*

Arguments

name-of-table

specifies a name for the error table. The default name for ErrorTable2 is *tname_UV*, where *tname* is the name of the target table. Table names exceeding 27 characters are truncated to accommodate the three-character suffix. Therefore, you might want to specify a name for the table that will not be truncated. When truncation occurs, a message is written to the log.

Restrictions This option is provided for the Load and Update operators. It is ignored by the Stream operator.

The name of an existing table cannot be used, unless you are restarting a paused job.

Details

ErrorTable2 contains records that violated the unique primary index constraint. This type of error occurs during the application phase of a Load or Update operator job.

The error table is created in the default user (logon) database, optionally qualified with a schema, unless the TD_LOGDB= table option is used to specify a different location for utility tables.

For information about the error table format and the procedure to correct errors, see *Teradata Parallel Transporter Reference*.

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)
- [“TD_LOGDB= Table Option” on page 485](#)

TD_LOG_MECH_TYPE= Table Option

Specifies the logon mechanism for a bulk load.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified within the “BULKOPTS= Table Option” on page 447
Data source:	Teradata

Syntax

TD_LOG_MECH_TYPE= *mechanism*

Arguments

mechanism

specifies the logon authentication mechanism. Currently, the valid value is LDAP.

Requirement The driver requires an 8-character input value. To submit, pad the value with four spaces as follows:

```
TD_LOG_MECH_TYPE="LDAP    "
```

Details

If the user connected via LDAP with the UID connection option, there is no reason to use this option. Use TD_LOG_MECH_TYPE= and TD_LOG_MECH_DATA= to allow LDAP authentication for TPT if the user did not specify LDAP authentication for UID in the connect string. You can also use this option with TD_LOG_MECH_DATA= to override the LDAP ID specified in the UID option with a different LDAP ID.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 446

TD_LOG_MECH_DATA= Table Option

Specifies additional data for the logon mechanism.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified within the “BULKOPTS= Table Option” on page 447

Data source: Teradata

Syntax

TD_LOG_MECH_DATA="*string*"

Arguments

string

used in conjunction with the TD_LOG_MECH_TYPE= table option, specifies credentials for authentication. Currently, only LDAP credentials are accepted. The credentials must be in the following form:

```
TD_LOG_MECH_DATA="authcid=value password=value realm=value"
```

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)
- [“TD_LOG_MECH_TYPE= Table Option” on page 483](#)

TD_LOG_TABLE= Table Option

Specifies the name of the restart log table.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified within the [“BULKOPTS= Table Option” on page 447](#)

Data source: Teradata

Syntax

TD_LOG_TABLE= *name-of-table*

Arguments

name-of-table

specifies the name of the restart log table for the Load, Stream, and Update operators. The restart log table contains restart information. The default name for the restart log table is *tname_RS*, where *tname* is the name of the target table. Table names exceeding 30 characters are truncated to accommodate the three-character suffix. Therefore, you might want to specify a name for the table that will not be truncated. When truncation occurs, a message is written to the log.

Details

The restart log table is created in the user's default (logon) database, unless the TD_LOGDB= table option is used to specify a different location.

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)
- [“TD_DROP_LOG_TABLE= Table Option” on page 480](#)
- [“TD_LOGDB= Table Option” on page 485](#)
- [“TD_WORKING_DB= Table Option” on page 496](#)

TD_LOGDB= Table Option

Specifies the database where the TPT utility tables are created.

Category: Bulk Loading

Requirements:	Must follow BULKLOAD=YES Must be specified within the “ BULKOPTS= Table Option ” on page 447
Data source:	Teradata

Syntax

TD_LOGDB= *database-name*

Arguments

database-name

specifies the name of the database where the utility tables are to be created. If this option is not specified, the utility tables are created in the specified SCHEMA. If SCHEMA is not specified, the tables are created in the default user (logon) database. The utility tables include ErrorTable1, ErrorTable2, the restart log table, and the work table.

See Also

Table Options:

- “[BULKLOAD= Table Option](#)” on page 446

TD_MAX_SESSIONS= Table Option

Specifies the maximum number of logon sessions that TPT can acquire for a job.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified within the “ BULKOPTS= Table Option ” on page 447
Data source:	Teradata

Syntax

TD_MAX_SESSIONS= *integer*

Arguments

integer

specifies the maximum number of logon sessions that the Teradata Parallel Transporter can acquire for an operator job. The default value is four sessions, if a value is not specified. The maximum value cannot be more than the number of AMPs available. See your Teradata documentation for details.

Details

The TD_MAX_SESSIONS= value must be greater than zero. Specifying a value less than 1 causes the job to terminate.

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)

TD_MIN_SESSIONS= Table Option

Specifies the minimum number of sessions for TPT to acquire before a job starts.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified within the “BULKOPTS= Table Option” on page 447
Data source:	Teradata

Syntax

TD_MIN_SESSIONS= *integer*

Arguments

integer

specifies the minimum number of sessions required for TPT to start a job. The number is applied to the Load, Stream, and Update operators. The default is one session.

Details

The TD_MIN_SESSIONS= value must be greater than zero and less than or equal to the maximum number of sessions. Specifying a value less than 1 causes the active operator to terminate.

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)

TD_NOTIFY_LEVEL= Table Option

Specifies the level at which log events are recorded.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified with the “BULKOPTS= Table Option” on page 447
Data source:	Teradata

Syntax

TD_NOTIFY_LEVEL= *level*

Arguments

level

Specifies the level at which events are reported. Valid settings are any of the following:

OFF	No notification of events is provided (the default value).
LOW	Initialize, CLIV2/DBS error, Exit events only
MEDIUM	All events except Err1 and Err2
HIGH	All events.

Details

If you set **TD_NOTIFY_LEVEL=**, set **TD_NOTIFY_METHOD=** to indicate how you want the events reported.

See Also

Table Options:

- [“TD_NOTIFY_METHOD= Table Option” on page 488](#)

TD_NOTIFY_METHOD= Table Option

Specifies the method for reporting events.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified with the “BULKOPTS= Table Option” on page 447
Data source:	Teradata

Syntax

TD_NOTIFY_METHOD= *method*

Arguments

method

specifies the notify method to be used for reporting events. Valid settings are any of the following:

NONE	No notification of events is provided (the default value)
MSG	Sends events to a log. On Windows, the events are sent to an EventLog that can be viewed using the Event Viewer. On UNIX, the destination of events is specified in the /etc/syslog.conf file.
EXIT	Sends events to a user-defined notify exit routine.

Details

If you set TD_NOTIFY_METHOD=, set TD_NOTIFY_LEVEL= to indicate the types of events that you want reported.

See Also

Table Options:

- [“TD_NOTIFY_LEVEL= Table Option” on page 487](#)
- [“TD_NOTIFY_STRING= Table Option” on page 489](#)

TD_NOTIFY_STRING= Table Option

Defines a string that precedes all messages sent to the system log.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified with the “BULKOPTS= Table Option” on page 447
Data source:	Teradata

Syntax

TD_NOTIFY_STRING= *string*

Arguments

string

Is an optional, user-specified string that precedes all messages that are sent to the system log. This string is also sent to the user-defined notify exit routine. If the NotifyMethod is MSG, the maximum length is 16 bytes. If the NotifyMethod is EXIT, the maximum length is 80 bytes.

See Also

Table Options:

- [“TD_NOTIFY_METHOD= Table Option” on page 488](#)

TD_PACK= Table Option

Specifies the number of statements to pack into a multistatement request.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified within the “BULKOPTS= Table Option” on page 447
Data source:	Teradata

Syntax

TD_PACK= *number*

Arguments

number

specifies the number of statements to pack into a multistatement request. The default number is 20. The maximum number of statements allowed is 600.

Restriction This option is ignored by the Load and Update operators.

See Also

Table Options:

- “TD_PACK_MAXIMUM= Table Option” on page 490

TD_PACK_MAXIMUM= Table Option

Lets the Stream operator determine the pack factor for the current Stream job.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified within the “BULKOPTS= Table Option” on page 447

Data source: Teradata

Syntax

TD_PACK_MAXIMUM= YES | NO

Arguments

YES

triggers the Stream operator to dynamically determine the maximum pack factor for the current Stream job.

Restriction This option is ignored by the Load and Update operators.

NO

loads the number of statements indicated by the TD_PACK= option for multistatement requests.

See Also

Table Options:

- “TD_PACK= Table Option” on page 489

TD_PAUSE_ACQ= Table Option

Forces a pause between the acquisition phase and the application phase of a load job.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified with the “BULKOPTS= Table Option” on page 447
Data source:	Teradata

Syntax

TD_PAUSE_ACQ= YES | NO

Arguments

YES

specifies to pause the load job after the acquisition phase and skip the application phase.

Restriction This option is provided for the Load and Update operators. It is ignored by the Stream operator.

NO

distributes all of the rows that were sent to the Teradata database during the acquisition phase to their final destination on the AMPs. This is the default value.

TD_SESSION_QUERY_BAND= Table Option

Passes a string of user-specified name=value pairs for use by the TPT session.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified with the “BULKOPTS= Table Option” on page 447
Data source:	Teradata

Syntax

TD_SESSION_QUERY_BAND= *string*

Arguments

string

specifies one or more query band expressions for use by each TPT session generated for the job or process. The bands are specified as name=value pairs.

Details

The bands are passed to the database for use by the server for load balancing. They are also used as trigger mechanisms by Teradata Multi-System Manager or TASM (Teradata Active System Management).

TD_TENACITY_HOURS= Table Option

Specifies the amount of time the TPT operator continues trying to log on to the Teradata database.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified within the “BULKOPTS= Table Option” on page 447
Data source:	Teradata

Syntax

TD_TENACITY_HOURS= *integer*

Arguments

integer

specifies the number of hours the TPT operator should attempt to log on to the Teradata database, when the maximum number of server utility slots are already occupied by other jobs. The default value is 0 (zero), meaning the operator does not try again and the job fails. It is recommended that you set a value when the Load and Update operators are used.

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)

TD_TENACITY_SLEEP= Table Option

Specifies the amount of time the TPT operator pauses, before retrying to log on to the Teradata database.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified within the “BULKOPTS= Table Option” on page 447
Data source:	Teradata

Syntax

TD_TENACITY_SLEEP= *integer*

Arguments

integer

specifies the number of minutes that the TPT operator pauses before retrying a job. This option is activated only when TD_TENACITY_HOURS= is set to a value. The default is six minutes.

Details

The TD_TENACITY_SLEEP= value must be greater than zero. If you specify a value of less than 1, the TPT operator responds with an error message and terminates the job.

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)

TD_TPT_OPER= Table Option

Specifies the load operator that is used by the Teradata Parallel Transporter.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified with the [“BULKOPTS= Table Option” on page 447](#)

Data source: Teradata

Syntax

TD_TPT_OPER= *operator*

Arguments

operator

Specifies the load operator that the Teradata Parallel Transporter (TPT) uses to load data. Valid values are LOAD, STREAM, or UPDATE.

LOAD specifies the Load operator, which inserts into an empty table. This is the fastest of the load operators.

STREAM specifies the Stream operator. It can be used to insert rows into empty Teradata tables or append to existing tables. The Stream operator is used by default when the TD_TPT_OPER= table option is not specified.

UPDATE specifies the Update operator, which inserts into existing tables. The Update operator is faster than the Stream operator.

Details

When BULKLOAD=YES is specified, the Teradata Parallel Transporter (TPT) is used to load data. TPT jobs are run using operators. These are discrete object-oriented modules that perform specific extraction, loading, and updating processes.

To obtain the best performance when the Update operator is used, it is recommended that you drop all unique secondary indexes, foreign key references, and join indexes onto target tables before the load.

The Teradata server supports a maximum of 16 concurrent utility slots. The Stream operator does not take up a utility slot. The Load and Update operators do take up a slot.

See Also

Table Options:

- “[BULKLOAD= Table Option](#)” on page 446
- “[TD_PAUSE_ACQ= Table Option](#)” on page 491

TD_TRACE_LEVEL= and TD_TRACE_LEVEL_INF= Table Options

Specifies the trace levels for driver tracing. TD_TRACE_LEVEL sets the primary trace level. TD_TRACE_LEVEL_INF sets the secondary trace level.

Category: Bulk Loading

Requirements: Must follow BULKLOAD=YES
Must be specified within the “[BULKOPTS= Table Option](#)” on page 447

Data source: Teradata

Syntax

TD_TRACE_LEVEL= *trace-level*

Arguments

trace-level

specifies the type of diagnostic messages written by each instance of the driver to an external log file. The diagnostic trace function provides more detailed information (including the version number) in the log file to aid in problem tracking and diagnosis. The following trace levels are available:

TD_OFF	disables driver tracing. TD_OFF is the default setting for both driver tracing and infrastructure tracing. No external log file is produced unless this default is changed. Specifying TD_OFF for both driver tracing and infrastructure tracing is the same as disabling tracing.
--------	--

TD_OPER	activates the tracing function for driver-specific activities. The absence of any value for the PauseAcq attribute means that the Update driver job executes both the acquisition phase and the application phase without pausing. This distributes all of the rows that were sent to the Teradata database during the acquisition phase to their final destination on the AMPs.
---------	--

TD_OPER_CLI	activates the tracing function for CLIV2-related activities (interaction with the Teradata database).
TD_OPER_NOTIFY	activates the tracing function for activities related to the Notify feature.
TD_OPER_OPCOMMON	activates the tracing function for activities involving the opcommon library.
TD_OPER_ALL	activates the tracing function for all of the tracing activities.

Details

If the trace level is set to any value other than TD_OFF, an external log file is created for each instance of the driver.

The trace levels for infrastructure tracing should be used only when you are directed to by Teradata support. TD_OFF, which disables infrastructure tracing, should always be specified.

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)

TD_TRACE_OUTPUT= Table Option

Specifies the name of the external file used for trace messages.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified within the “BULKOPTS= Table Option” on page 447
Data source:	Teradata

Syntax

TD_TRACE_OUTPUT= *filename*

Arguments

filename

specifies the name of the external file to use for tracing. If a file with the specified name already exists, then the existing file is overwritten. The new filename is created with the name of the driver and a time stamp.

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)

TD_WORK_TABLE= Table Option

Specifies a name for the TPT Work table.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified with the “BULKOPTS= Table Option” on page 447
Data source:	Teradata

Syntax

TD_WORK_TABLE= *name-of-table*

Arguments

name-of-table

specifies a name for the TPT Work table. The default name for the table is *ttname_WT*, where *ttname* is the name of the target table. Table names exceeding 27 characters are truncated to accommodate the three-character suffix. Therefore, you might want to specify a name for the table that will not be truncated. When truncation occurs, a message is written to the log.

Restriction This option is provided for the Update operator. It is ignored by the Load and Stream operators.

Details

The Work table is created in the default user (logon) database, optionally qualified with a schema, unless the TD_LOGDB= table option is used to specify a different location for utility tables.

See Also

Table Options:

- [“BULKLOAD= Table Option” on page 446](#)
- [“TD_DROP_WORK_TABLE= Table Option” on page 480](#)
- [“TD_LOGDB= Table Option” on page 485](#)

TD_WORKING_DB= Table Option

Specifies the database where the insert table is to be created.

Category:	Bulk Loading
Requirements:	Must follow BULKLOAD=YES Must be specified with the “BULKOPTS= Table Option” on page 447
Data source:	Teradata

Syntax

TD_WORKING_DB= *database-name*

Arguments

database-name

specifies a database name. If this option is not specified, the table is created in the specified SCHEMA. If SCHEMA is not specified, the table is created in the default user (logon) database.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 446

THREADNUM= Table Option

Specifies the maximum number of I/O threads the SPD Engine can spawn for processing an SPD Engine data set.

Category: Table Control

Data source: SPD Engine data set

Syntax

THREADNUM= *n*

Arguments

n

specifies the number of threads. The default is the value of the SPDEMAXTHREADS= system option, if set. Otherwise, the default is two times the number of CPUs on your computer.

Details

THREADNUM= enables you to specify the maximum number of I/O threads that the SPD Engine spawns for processing an SPD Engine data set. The THREADNUM= value applies to any of the following SPD Engine I/O processing:

- WHERE expression processing
- parallel index creation
- I/O requested by thread-enabled applications

Adjusting THREADNUM= enables the system administrator to adjust the level of CPU resources the SPD Engine can use for any process. For example, in a 64-bit processor system, setting THREADNUM=4 limits the process to, at most, four CPUs, thereby enabling greater throughput for other users or applications.

When THREADNUM= is greater than 1, parallel processing is likely to occur. Therefore, physical order might not be retained in the output. Setting THREADNUM=1 means that no parallel processing occurs

You can also use this option to explore scalability for WHERE expression evaluations.

SPDEMAXTHREADS=, a configurable system option, imposes an upper limit on the consumption of system resources and therefore constrains the THREADNUM= value.

TYPE= Table Option

Specifies the data set type for a specially structured SAS data set.

Category: Table Control

Data source: SAS data set, SPD Engine data set

Syntax

TYPE= *data-set-type*

Arguments

data-set-type

specifies the special type of the data set.

Details

Use the TYPE= table option to create a special data set in the proper format or to identify the special type of the data set.

Most data sets do not have a specified type. However, there are several specially structured data sets that are used by some SAS/STAT procedures. These SAS data sets contain special columns and rows, and they are usually created by SAS statistical procedures. Because most of the special data sets are used with SAS/STAT software, they are described in *SAS/STAT User's Guide*.

Other values are available in other SAS software products and are described in the appropriate documentation.

UCA= Table Option

Specifies to use Unicode Collation Algorithms (UCA) collation to determine the ordering of character columns in the ORDERBY= option.

Category: Table Control

Default: NO

Interaction: Must follow ORDERBY=.

Data source: SASHDAT file

Syntax

PARTITION=(*key*) ORDERBY=(*column-list*) UCA=YES | NO

See Also

Table Options:

- [“ORDERBY= Table Option”](#) on page 466

UNIQUESAVE= Table Option

Specifies to save observations with nonunique key values (the rejected observations) to a separate data set when inserting observations into data sets with unique indexes.

Valid in: INSERT statement

Category: User Control of SAS Index Usage

Interaction: Used in conjunction with SPDSUSDS= automatic macro variable

Data source: SPD Engine data set

Syntax

UNIQUESAVE= [YES](#) | [NO](#)

Arguments

YES

writes rejected observations to a separate, system-created table that can be accessed by a reference to the macro variable SPDSUSDS=.

NO

does not write rejected observations to a separate table (that is, ignores nonunique key values).

Details

When observations are inserted into a data set that has a unique index, the rejected observations are ignored. With UNIQUESAVE=YES, the rejected observations are saved to a separate data set whose name is stored in the macro variable SPDSUSDS. You can specify the macro variable in place of the data set name to identify the rejected observations.

WHEREINDEX= Table Option

Specifies a list of indexes to exclude when making WHERE expression evaluations.

Category: User Control of SAS Index Usage

Restriction: Cannot be used with IDXWHERE=NO

Data source: SPD Engine data set

Syntax

WHEREINDEX= ([name1](#) [name2](#)...)

Arguments**(name1 name2...)**

specifies a list of index names that you want to exclude from use.

See Also**Table Options:**

- [“IDXWHERE= Table Option” on page 462](#)

WRITE= Table Option

Assigns a WRITE password to a SAS data set or SPD Engine data set that prevents users from writing to the file or that enables access to a Write-protected file.

Category: Table Control**Data source:** SAS data set, SPD Engine data set

Note: Check your log after this operation to ensure that the password values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in *SAS Language Reference: Concepts*.

Syntax**WRITE=** *write-password***Arguments*****write-password***

must be a valid SAS name.

Details

You can use this option to assign a password or to access a write-protected file.

Note: A SAS password does not control access to a SAS file beyond the SAS system.

You should use the operating system-supplied utilities and file-system security controls in order to control access to SAS files outside of SAS.

Part 3

Appendixes

<i>Appendix 1</i>	
FedSQL and the ANSI Standard	503
<i>Appendix 2</i>	
Data Type Reference	507
<i>Appendix 3</i>	
Using FedSQL and DS2	535
<i>Appendix 4</i>	
Tables Used in Examples	537
<i>Appendix 5</i>	
DICTIONARY Table Descriptions	545
<i>Appendix 6</i>	
Usage Notes	557
<i>Appendix 7</i>	
ICU License	559

Appendix 1

FedSQL and the ANSI Standard

Compliance	503
FedSQL Enhancements	503
Reserved Words	503
Column Modifiers	503
Functions	504
Table Options	504
SAS Missing Values and Null Values	504
DS2 Package Methods as Functions	504
Integrity Constraints	504
FedSQL DICTIONARY Tables	505
FedSQL Limitations	505
Granting User Privileges	505
Base SAS and SPD Engine Identifiers and Naming Conventions	505

Compliance

FedSQL is core compliant with the 1999 ANSI Standard for SQL.¹

FedSQL Enhancements

Reserved Words

For a complete list of reserved words in FedSQL, see [“FedSQL Reserved Words”](#) on [page 61](#).

Column Modifiers

FedSQL supports the FORMAT=, INFORMAT=, and LABEL= modifiers for expressions within the SELECT clause. These modifiers control the format in which output data is displayed and labeled. For more information, see [“How to Store, Change, Delete, and Use Stored Formats ”](#) on [page 70](#), [“How to Specify Informats in FedSQL”](#) on [page 347](#), and the [“HAVING Clause”](#) on [page 401](#).

¹ International Organization for Standardization (ISO): Document ISO/IEC 9075–1:1999. Also available as American National Standards Institute (ANSI) Document ANSI 9075–1:1999.

Functions

Aggregate Functions

FedSQL supports many more aggregate functions that required by the ANSI Standard for SQL. For a complete list of aggregate functions that FedSQL supports, see [“Aggregate Functions” on page 182](#).

Base SAS DATA Step Functions

FedSQL supports many of the functions that are available to the Base SAS DATA step. Functions that are not supported include the variable information functions and functions that work with arrays of data. Other FedSQL data sources support their own sets of functions.

For a complete list of functions that FedSQL supports, see [“Overview of FedSQL Functions” on page 179](#).

PROC FCMP Functions

FedSQL supports user-written functions, except those functions with array elements, only when FedSQL statements are submitted with PROC FEDSQL. For more information about using PROC FEDSQL, see *Base SAS Procedures Guide*.

Table Options

A table option specifies actions that enable you to perform operations on a table such as assigning or specifying passwords. A table option in the FedSQL performs much of the same functionality as a Base SAS data set option. For more information, see [“Overview of Statement Table Options” on page 417](#).

SAS Missing Values and Null Values

FedSQL supports both SAS missing values and null values. Nonexistent data is represented by a SAS missing value in SAS data sets and in SPD Engine data sets. For all other data sources, nonexistent data is represented by an ANSI SQL null value. For more information, see [“How FedSQL Processes Nulls and SAS Missing Values” on page 18](#).

DS2 Package Methods as Functions

You can invoke a DS2 package method as a function in a FedSQL SELECT statement. For more information, see [“Using DS2 Packages in Expressions” on page 181](#).

Integrity Constraints

Integrity constraints are a set of data validation rules that you can specify to preserve the validity and consistency of your data. Integrity constraints that are specified in the CREATE TABLE statement are passed through to the data source. When a transaction modifies the table, the data source enforces integrity constraints. For more information, see the [“CREATE TABLE Statement” on page 359](#).

FedSQL DICTIONARY Tables

FedSQL supports DICTIONARY tables that are similar to the DICTIONARY tables that are available in Base SAS. A FedSQL DICTIONARY table is a Read-only table that contains information about columns, tables, and catalogs, and statistics about a single table and its associated indexes. For more information, see [“DICTIONARY Tables” on page 58](#).

FedSQL Limitations

Granting User Privileges

The GRANT statement, PRIVILEGES keyword, and authorization-identifier features of SQL are not supported. You might want to use operating environment-specific means of security instead.

Base SAS and SPD Engine Identifiers and Naming Conventions

Table, column, and index names for SAS data sets and SPD Engine data sets are limited to 32 characters. For more information, see [“Identifiers” on page 17](#).

Appendix 2

Data Type Reference

Data Types for SAS Data Sets	507
Data Types for SPD Engine Data Sets	509
Data Types for Aster	511
Data Types for DB2 under UNIX and PC Hosts	512
Data Types for Greenplum	513
Data Types for HDMD	514
Data Types for Hive	516
Data Types for MDS	517
Data Types for MySQL	519
Data Types for Netezza	521
Data Types for ODBC	522
Data Types for Oracle	523
Data Types for PostgreSQL	525
Data Types for SAP	527
Data Types for SAP HANA	529
Data Types for SASHDAT	530
Data Types for Sybase IQ	531
Data Types for Teradata	533

Data Types for SAS Data Sets

The following table lists the data type support for a SAS data set.

The BINARY and VARBINARY data types are not supported for data type definition.

For some data type definitions, the data type is mapped to CHAR, which is a Base SAS character data type, or DOUBLE, which is a Base SAS numeric data type. For data source-specific information about the SAS numeric and SAS character data types, see *SAS Language Reference: Concepts*.

Table A2.1 Data Types for SAS Data Sets

Data Type Definition Keyword*	SAS Data Set Data Type	Description	Data Type Returned
BIGINT**	DOUBLE	64-bit double precision, floating-point number. <i>Note:</i> There is potential for loss of precision.	DOUBLE
CHAR(<i>n</i>)	CHAR(<i>n</i>)	Fixed-length character string. <i>Note:</i> Cannot contain ANSI SQL null values.	CHAR(<i>n</i>)
DATE ***	DOUBLE	64-bit double precision, floating-point number. By default, applies the DATE9 SAS format.	DOUBLE
DECIMAL NUMERIC(<i>p,s</i>)**	DOUBLE	64-bit double precision, floating-point number.	DOUBLE
DOUBLE**	DOUBLE	64-bit double precision, floating-point number.	DOUBLE
FLOAT(<i>p</i>)**	DOUBLE	64-bit double precision, floating-point number.	DOUBLE
INTEGER**	DOUBLE	64-bit double precision, floating-point number.	DOUBLE
NCHAR(<i>n</i>)	CHAR(<i>n</i>)	Fixed-length character string. By default, sets the encoding to Unicode UTF-8. †	CHAR(<i>n</i>)
NVARCHAR(<i>n</i>)	CHAR(<i>n</i>)	Fixed-length character string. By default, sets the encoding to Unicode UTF-8. †	CHAR(<i>n</i>)
REAL**	DOUBLE	64-bit double precision, floating-point number.	DOUBLE
SMALLINT**	DOUBLE	64-bit double precision, floating-point number.	DOUBLE
TIME(<i>p</i>)***	DOUBLE	64-bit double precision, floating-point number. By default, applies the TIME8 SAS format.	DOUBLE

Data Type Definition Keyword*	SAS Data Set Data Type	Description	Data Type Returned
TIMESTAMP(<i>p</i>)***	DOUBLE	64-bit double precision, floating-point number. By default, applies the DATETIME19.2 SAS format.	DOUBLE
TINYINT**	DOUBLE	64-bit double precision, floating-point number.	DOUBLE
VARCHAR(<i>n</i>)	CHAR(<i>n</i>)	Fixed-length character string. <i>Note:</i> Cannot contain ANSI SQL null values.	CHAR(<i>n</i>)

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** Do not apply date and time SAS formats to a numeric data type. For date and time values, use the DATE, TIME, or TIMESTAMP data types.

*** Because the values are stored as a double precision, floating-point number, you can use the values in arithmetic expressions.

† UTF-8 is an MBCS encoding. Depending on the operating environment, UTF-8 characters are of varying width, from one to four bytes. The value for *n*, which is the maximum number of multibyte characters to store, is multiplied by the maximum length for the operating environment. Note that when you are transcoding, such as from UTF-8 to Wlatin2, the variable lengths (in bytes) might not be sufficient to hold the values, and the result is character data truncation.

Data Types for SPD Engine Data Sets

The following table lists the data type support for an SPD Engine data set.

The BINARY, DECIMAL, NUMERIC, NCHAR, NVARCHAR, and VARBINARY data types are not supported for data type definition.

For some data type definitions, the data type is mapped to CHAR, which is a Base SAS character data type, or DOUBLE, which is a Base SAS numeric data type. For data source specific information about the SAS numeric and SAS character data types, see *SAS Language Reference: Concepts*.

Table A2.2 Data Types for SPD Engine Data Sets

Data Type Definition Keyword	SPD Data Set Data Type	Description	Data Type Returned
BIGINT*	DOUBLE	64-bit double precision, floating-point number. <i>Note:</i> There is potential for loss of precision.	DOUBLE

Data Type Definition Keyword	SPD Data Set Data Type	Description	Data Type Returned
CHAR(<i>n</i>)	CHAR(<i>n</i>)	Fixed-length character string. <i>Note:</i> Cannot contain ANSI SQL null values.	CHAR(<i>n</i>)
DATE **	DOUBLE	64-bit double precision, floating-point number. By default, applies the DATE9 SAS format.	DOUBLE
DOUBLE*	DOUBLE	64-bit double precision, floating-point number.	DOUBLE
FLOAT(<i>p</i>)*	DOUBLE	64-bit double precision, floating-point number.	DOUBLE
INTEGER*	DOUBLE	64-bit double precision, floating-point number.	DOUBLE
REAL*	DOUBLE	64-bit double precision, floating-point number.	DOUBLE
SMALLINT*	DOUBLE	64-bit double precision, floating-point number.	DOUBLE
TIME(<i>p</i>)**	DOUBLE	64-bit double precision, floating-point number. By default, applies the TIME8 SAS format.	DOUBLE
TIMESTAMP(<i>p</i>)**	DOUBLE	64-bit double precision, floating-point number. By default, applies the DATETIME19.2 SAS format.	DOUBLE
TINYINT*	DOUBLE	64-bit double precision, floating-point number.	DOUBLE
VARCHAR(<i>n</i>)	CHAR(<i>n</i>)	Fixed-length character string. <i>Note:</i> Cannot contain ANSI SQL null values.	CHAR(<i>n</i>)

* Do not apply date and time SAS formats to a numeric data type. For date and time values, use DATE, TIME, or TIMESTAMP data types.

** Because the values are stored as double precision, floating-point numbers, you can use the values in arithmetic expressions.

Data Types for Aster

The following table lists the data type support for an Aster database.

The NCHAR, NVARCHAR, and TINYINT data types are not supported for data type definition.

For data source specific information about Aster database data types, see the Aster database documentation.

Table A2.3 Data Types for Aster

Data Type Definition Keyword	Aster Data Type	Description	Data Type Returned
BIGINT	BIGINT	Large signed, exact whole number.	BIGINT
BINARY(<i>n</i>)	BYTEA	Varying-length binary string.	BINARY(<i>n</i>)
*	BOOL	One byte integral data type that can contain values 0, 1, or NULL.	*
CHAR(<i>n</i>)	CHAR(<i>n</i>)	Fixed-length character string.	CHAR(<i>n</i>)
DATE	DATE	Date values.	DATE
DECIMAL NUMERIC(<i>p,s</i>)	NUMERIC(<i>p,s</i>)	Signed, fixed-point decimal number.	DECIMAL NUMERIC(<i>p,s</i>)
DOUBLE	FLOAT(<i>p</i>)	Signed, double precision, floating-point number.	DOUBLE
FLOAT(<i>p</i>)	FLOAT(<i>p</i>)	Signed, double precision, floating-point number.	FLOAT(<i>p</i>)
INTEGER	INTEGER	Regular signed, exact whole number.	INTEGER
REAL	REAL	Signed, single precision, floating-point number.	REAL
SMALLINT	SMALLINT	Small signed, exact whole number.	SMALLINT
TIME(<i>p</i>)	TIME(<i>p</i>)	Time value.	TIME(<i>p</i>)
TIMESTAMP(<i>p</i>)	TIMESTAMP(<i>p</i>)	Date and time value.	TIMESTAMP(<i>p</i>)

Data Type Definition Keyword	Aster Data Type	Description	Data Type Returned
VARBINARY(<i>n</i>)	BYTEA	Varying-length binary string.	VARBINARY(<i>n</i>)
*	TEXT	Varying-length large character string.	VARCHAR(<i>n</i>)
VARCHAR(<i>n</i>)	VARCHAR(<i>n</i>)	Varying-length character string.	VARCHAR(<i>n</i>)

* The Aster data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

Data Types for DB2 under UNIX and PC Hosts

The following table lists the data type support for a DB2 database under UNIX and PC hosts.

The NCHAR, NVARCHAR, and TINYINT data types are not supported for data type definition.

For data source specific information about the DB2 database data types, see the DB2 database documentation.

Table A2.4 Data Types for DB2 under UNIX and PC Hosts

Data Type Definition Keyword*	DB2 Data Type	Description	Data Type Returned
BIGINT	BIGINT	Large signed, exact whole number.	BIGINT
BINARY(<i>n</i>)	CHAR(<i>n</i>) FOR BIT DATA	Fixed-length binary string.	BINARY(<i>n</i>)
**	BLOB(<i>n</i> [K M G])	Varying-length binary large object string.	**
CHAR(<i>n</i>)	CHAR(<i>n</i>)	Fixed-length character string.	CHAR(<i>n</i>)
**	CLOB(<i>n</i> [K M G])	Varying-length character large object string.	**
DATE	DATE	Date values.	DATE
**	DBCLOB(<i>n</i> [K M G])	Varying-length double-byte character large object.	**
DECIMAL NUMERIC(<i>p,s</i>)	DECIMAL NUMERIC(<i>p,s</i>)	Signed, fixed-point decimal number.	DECIMAL NUMERIC(<i>p,s</i>)

Data Type Definition Keyword*	DB2 Data Type	Description	Data Type Returned
DOUBLE	DOUBLE	Signed, double precision, floating-point number.	DOUBLE
FLOAT(<i>p</i>)	FLOAT(<i>p</i>)	Signed, single precision or double precision, floating-point number.	FLOAT(<i>p</i>)
**	GRAPHIC(<i>n</i>)	Fixed-length graphic string.	**
INTEGER	INTEGER	Regular signed, exact whole number.	INTEGER
**	LONG VARCHAR [FOR BIT DATA]	Varying-length character or binary string.	**
**	LONG VARGRAPHIC(<i>n</i>)	Varying-length graphic string.	**
REAL	REAL	Signed, single precision, floating-point number.	REAL
SMALLINT	SMALLINT	Small signed, exact whole number.	SMALLINT
TIME(<i>p</i>)	TIME(<i>p</i>)	Time value.	TIME(<i>p</i>)
TIMESTAMP(<i>p</i>)	TIMESTAMP(<i>p</i>)	Date and time value.	TIMESTAMP(<i>p</i>)
VARBINARY(<i>n</i>)	VARCHAR(<i>n</i>) FOR BIT DATA	Varying-length binary string.	VARBINARY(<i>n</i>)
VARCHAR(<i>n</i>)	VARCHAR(<i>n</i>)	Varying-length character string.	VARCHAR(<i>n</i>)
**	VARGRAPHIC(<i>n</i>)	Varying-length graphic string	**

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** The DB2 data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

Data Types for Greenplum

The following table lists the data type support for a Greenplum database.

The BINARY, NCHAR, NVARCHAR, and TINYINT data types are not supported for data type definition.

For data source specific information about Greenplum data types, see the Greenplum database documentation.

Table A2.5 Data Types for Greenplum

Data Type Definition Keyword*	Greenplum Data Type	Description	Data Type Returned
BIGINT	INT8	Large signed, exact whole number.	BIGINT
CHAR(<i>n</i>)	CHAR(<i>n</i>)	Fixed-length character string.	CHAR(<i>n</i>)
DATE	DATE	Date values.	DATE
DECIMAL NUMERIC(<i>p,s</i>)	DECIMAL(<i>p,s</i>)	Signed, fixed-point decimal number.	DECIMAL NUMERIC(<i>p,s</i>)
DOUBLE	DOUBLE	Floating-point number.	DOUBLE
FLOAT(<i>p</i>)	FLOAT8(<i>p</i>)	Floating-point number.	FLOAT(<i>p</i>)
INTEGER	INTEGER	Regular signed, exact whole number.	INTEGER
REAL	REAL	Floating-point number.	REAL
SMALLINT	INT8	Small signed, exact whole number.	SMALLINT
TIME(<i>p</i>)	TIME(<i>p</i>)	Time value in hours, minutes, and seconds.	TIME(<i>p</i>)
TIMESTAMP(<i>p</i>)	TIMESTAMP(<i>p</i>)	Date and time value.	TIMESTAMP(<i>p</i>)
VARBINARY(<i>n</i>)	BYTEA	Varying-length binary string.	VARBINARY(<i>n</i>)
VARCHAR(<i>n</i>)	VARCHAR(<i>n</i>)	Varying-length character string.	VARCHAR(<i>n</i>)

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

Data Types for HDMD

The following table lists the data type support for HDMD.

The BINARY, DECIMAL/NUMERIC, FLOAT, NCHAR, and NVARCHAR data types are not supported for data type definition.

Table A2.6 Data Types for HDMD

Data Type Definition Keyword **	HDMD Data Type	Description	Data Type Returned
BIGINT	BIGINT	A large signed, exact whole number, with a precision of 19 digits. The range of integers is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.	BIGINT
CHAR(<i>n</i>)	CHAR(<i>n</i>)	A fixed-length character string.	CHAR(<i>n</i>)
DATE	DATE	Date value.	DATE
DOUBLE	DOUBLE	A signed, approximate, double-precision, floating-point number. <i>Note:</i> Supports SAS missing values	DOUBLE
INTEGER	INTEGER	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
REAL	REAL	A signed, approximate, single-precision, floating-point number. <i>Note:</i> Supports SAS missing values	REAL
SMALLINT	SMALLINT	A small signed, exact whole number, with a precision of five digits. The range of integers is -32,768 to 32,767.	INTEGER
TIME(<i>p</i>)	TIME[<i>p</i>]	Time value with optional precision.	TIME[(<i>p</i>)]
TIMESTAMP(<i>p</i>)	TIMESTAMP[<i>p</i>]	Date and time value with optional precision.	TIMESTAMP[(<i>p</i>)]

Data Type Definition Keyword **	HDMD Data Type	Description	Data Type Returned
TINYINT	TINYINT	A very small signed, exact whole number, with a precision of three digits. The range of integers is -128 to 127.	INTEGER
VARCHAR(<i>n</i>)	VARCHAR(<i>n</i>)	Varying-length character string data.	VARCHAR(<i>n</i>)

Data Types for Hive

The following table lists the data type support for Hive. Hive versions 0.10 and later are supported.

The DECIMAL(*p,s*), NCHAR, NVARCHAR, and VARBINARY data types are not available for data definition.

Hive complex types are not supported.

For data-source specific information about Hive data types, see the Hive database documentation.

Table A2.7 Data Types for Hive

Data Type Definition Keyword	Hive Data Type	Description	Data Type Returned
BIGINT	BIGINT	A signed eight-byte integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.	BIGINT
BINARY(<i>n</i>)	BINARY	A varying length binary string.	BINARY
*	BOOLEAN	A textual true or false value.	TINYINT
CHAR(<i>n</i>)	CHAR(<i>n</i>)**	A character string up to 255 characters.	CHAR
DATE	DATE***†	An ANSI SQL date type.	DATE
DECIMAL/NUMERIC(<i>p,s</i>)	DECIMAL	A fixed-point decimal number, with 38 digits precision.	DOUBLE

Data Type Definition Keyword	Hive Data Type	Description	Data Type Returned
DOUBLE	DOUBLE	An eight-byte, double-precision floating-point number.	DOUBLE
FLOAT(<i>p</i>)	FLOAT(<i>p</i>)	A four-byte, single-precision floating-point number.	FLOAT(<i>p</i>)
INTEGER	INTEGER	A signed four-byte integer.	INTEGER
REAL	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
SMALLINT	SMALLINT	A signed two-byte integer, from -32,768 to 32,767.	SMALLINT
*	STRING	A variable-length character string.	VARCHAR(<i>n</i>) ^{††††}
TIME(<i>p</i>)		A time value.	STRING [‡]
TIMESTAMP(<i>p</i>)	TIMESTAMP	A UNIX timestamp with optional nanosecond precision.	TIMESTAMP[(<i>p</i>)]
TINYINT	TINYINT	A signed one-byte integer, from -128 to 127.	TINYINT
VARCHAR(<i>n</i>)	VARCHAR(<i>n</i>)	A varying-length character string.	VARCHAR(<i>n</i>)

* The Hive data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

** Full support for this data type is available in Hive 0.13 and later. In Hadoop environments that use earlier Hive versions (which do not support the CHAR type), columns defined as CHAR are mapped to VARCHAR.

*** Full support for this data type is available in Hive 0.12 and later. In Hadoop environments that use earlier Hive versions (which do not support the DATE type), any SASFMT TableProperties that are defined on STRING columns are applied when reading Hive, effectively allowing the STRING columns to be treated as DATE columns. When the DATE data type is used for data definition in earlier Hive versions, the DATE type is mapped to a STRING column with SASFMT TableProperties. For more information about SASFMT TableProperties, see “SAS Table Properties for Hive and HADOOP” in *SAS/ACCESS for Relational Databases: Reference*.

† Hive has a maximum year value of 9999. Date values containing higher years will be read back as null values.

†† The maximum length of VARCHAR(*n*) is determined by the DBMAX_TEXT= data source connection option.

††† SASFMT Table Properties are applied when reading STRING columns.

‡ Hive does not support the TIME(*p*) data type. When data is being read from Hive, STRING columns that have SASFMT TableProperties defined that specify the SAS DATE8. format are converted to the TIME(*p*) data type. When the TIME type is used for data definition, it is mapped to a STRING column with SASFMT TableProperties. For more information about SASFMT TableProperties, see “SAS Table Properties for Hive and HADOOP” in *SAS/ACCESS for Relational Databases: Reference*.

Data Types for MDS

The following table lists the data type support for the in-memory database.

Table A2.8 Data Types for MDS

Data Type Definition Keyword **	MDS Data Type	Description	Data Type Returned
BIGINT	BIGINT	64-bit, signed integer.	BIGINT
BINARY(<i>n</i>)	BINARY(<i>n</i>)	Fixed-length binary data.	BINARY(<i>n</i>)
CHAR(<i>n</i>)	CHAR(<i>n</i>)	Fixed-length character string data.	CHAR(<i>n</i>)
DATE	DATE	Date value.	DATE
DECIMAL NUMERIC(<i>p,s</i>)	NUMERIC(<i>p,s</i>)	Precision scale numeric.	DECIMAL/NUMERIC(<i>p,s</i>)
DOUBLE	DOUBLE	8-byte IEEE floating-point value. <i>Note:</i> Supports ANSI SQL null values	DOUBLE
FLOAT(<i>p</i>)	DOUBLE	8-byte IEEE floating-point value. <i>Note:</i> Supports ANSI SQL null values	DOUBLE
INTEGER	INTEGER	32-bit, signed integer.	INTEGER
NCHAR(<i>n</i>)	NCHAR(<i>n</i>)	Fixed-length Unicode character string.	NCHAR(<i>n</i>)
NVARCHAR(<i>n</i>)	NVARCHAR(<i>n</i>)	Varying-length Unicode character string.	NVARCHAR(<i>n</i>)
REAL	DOUBLE	8-byte IEEE floating-point value. <i>Note:</i> Supports ANSI SQL null values	DOUBLE
SMALLINT	INTEGER	32-bit, signed integer.	INTEGER
TIME(<i>p</i>)	TIME(<i>p</i>)	Time value.	TIME(<i>p</i>)
TIMESTAMP(<i>p</i>)	TIMESTAMP(<i>p</i>)	Date and time value.	TIMESTAMP(<i>p</i>)
TINYINT	INTEGER	32-bit, signed integer.	INTEGER
*	UBIGINT	64-bit, unsigned integer.	BIGINT
*	UINTeger	32-bit, unsigned integer.	INTEGER
VARCHAR(<i>n</i>)	VARCHAR(<i>n</i>)	Varying-length character string data.	VARCHAR(<i>n</i>)

Data Type Definition Keyword **	MDS Data Type	Description	Data Type Returned
VARBINARY(n)	VARBINARY(n)	Varying-length binary data.	VARBINARY(n)

* The MDS SQL data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

** The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

Data Types for MySQL

The following table lists the data type support for a MySQL database.

The NCHAR, NVARCHAR, REAL, and VABINARY data types are not supported for data type definition.

For data source specific information about MySQL data types, see the MySQL database documentation.

Table A2.9 Data Types for MySQL

Data Type Definition Keyword*	MySQL Data Type	Description	Data Type Returned
BIGINT	BIGINT	Large signed, exact whole number.	BIGINT
BINARY(n)	BINARY(n)	Fixed-length binary string.	BINARY(n)
**	BLOB	Varying-length binary large object string.	**
CHAR(n)	CHAR(n)	Fixed-length character string.	CHAR(n)
DATE	DATE	Date values.	DATE
**	DATETIME	Date and time value.	**
DECIMAL NUMERIC(p,s)	DECIMAL(p,s)	Signed, fixed-point decimal number.	DECIMAL(p,s)
DOUBLE	DOUBLE	Signed, double precision, floating-point number.	DOUBLE
**	ENUM(values)	Character values from a list of allowed values.	**

Data Type Definition Keyword*	MySQL Data Type	Description	Data Type Returned
FLOAT(<i>p</i>)	FLOAT(<i>p</i>)	Signed, single precision or double precision, floating-point number.	FLOAT(<i>p</i>)
INTEGER	INT	Regular signed, exact whole number.	INTEGER
**	LOBLOB	Varying-length binary data.	**
**	LONGTEXT	Varying-length character string.	**
**	MEDIUMBLOB	Varying-length binary data.	**
**	MEDIUMINT	Regular signed, exact whole number.	**
**	MEDIUMTEXT	Varying-length character string.	**
**	SET(<i>values</i>)	Character values from a list of allowed values.	**
SMALLINT	SMALLINT	Small signed, exact whole number.	SMALLINT
**	TEXT	Varying-length text data.	**
TIME(<i>p</i>)	TIME(<i>p</i>)	Time value.	TIME(<i>p</i>)
TIMESTAMP(<i>p</i>)	TIMESTAMP(<i>p</i>)	Date and time value.	TIMESTAMP(<i>p</i>)
**	TINYBLOB	Varying-length binary large object string.	**
TINYINT	TINYINT	Very small signed, exact whole number.	TINYINT
**	TINYTEXT	Varying-length text data.	**
VARCHAR(<i>n</i>)	VARCHAR(<i>n</i>)	Varying-length character string.	VARCHAR(<i>n</i>)

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** The MySQL data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

Data Types for Netezza

The following table lists the data type support for a Netezza database.

The BINARY and VARBINARY data types are not supported for data type definition.

For data source specific information about Netezza data types, see the Netezza database documentation.

Table A2.10 Data Types for Netezza

Data Type Definition Keyword*	Netezza Data Type	Description	Data Type Returned
BIGINT	BIGINT	Large signed, exact whole number.	BIGINT
CHAR(<i>n</i>)	CHAR(<i>n</i>)	Fixed-length character string data.	CHAR(<i>n</i>)
DATE	DATE	Date values.	DATE
DECIMAL NUMERIC(<i>p,s</i>)	DECIMAL(<i>p,s</i>)	Fixed-point decimal number.	DECIMAL(<i>p,s</i>)
DOUBLE	DOUBLE	Floating-point number.	DOUBLE
FLOAT(<i>p</i>)	FLOAT(<i>p</i>)	64-bit double precision, floating-point number.	FLOAT(<i>p</i>)
INTEGER	INTEGER	Large integer.	INTEGER
NCHAR(<i>n</i>)	NCHAR(<i>n</i>)	Fixed-length Unicode character string.	NCHAR(<i>n</i>)
NVARCHAR(<i>n</i>)	NVARCHAR(<i>n</i>)	Varying-length Unicode character string.	NVARCHAR(<i>n</i>)
REAL	REAL	Floating-point number.	REAL
SMALLINT	SMALLINT	Small integer.	SMALLINT
TIME(<i>p</i>)	TIME(<i>p</i>)	Time value.	TIME(<i>p</i>)
TIMESTAMP(<i>p</i>)	TIMESTAMP(<i>p</i>)	Date and time value.	TIMESTAMP(<i>p</i>)
TINYINT	BYTEINT	Tiny integer.	TINYINT

Data Type Definition Keyword*	Netezza Data Type	Description	Data Type Returned
VARCHAR(<i>n</i>)	VARCHAR(<i>n</i>)	Varying-length character string data.	VARCHAR(<i>n</i>)

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

Data Types for ODBC

The following table lists the data type support for an ODBC-compliant data source.

For data source specific information about ODBC SQL data types, see the specific ODBC data source documentation.

Table A2.11 Data Types for ODBC

Data Type Definition Keyword*	ODBC SQL Identifier	Description	Data Type Returned
BIGINT	SQL_BIGINT	Large signed, exact whole number.	BIGINT
BINARY(<i>n</i>)	SQL_BINARY	Fixed-length binary string.	BINARY(<i>n</i>)
**	SQL_BIT	Single bit binary data.	**
CHAR(<i>n</i>)	SQL_CHAR	Fixed-length character string.	CHAR(<i>n</i>)
DATE	SQL_TYPE_DATE	Date values.	DATE
DECIMAL NUMERIC(<i>p,s</i>)	SQL_DECIMAL SQL_NUMERIC	Signed, fixed-point decimal number.	DECIMAL NUMERIC(<i>p,s</i>)
DOUBLE	SQL_DOUBLE	Signed, double precision, floating-point number.	DOUBLE
FLOAT(<i>p</i>)	SQL_FLOAT	Signed, approximate, floating-point number.	FLOAT(<i>p</i>)
**	SQL_GUID	Globally unique identifier.	**
INTEGER	SQL_INTEGER	Regular signed, exact whole number.	INTEGER
**	SQL_INTERVAL	Intervals between two years, months, days, dates or times.	**

Data Type Definition Keyword*	ODBC SQL Identifier	Description	Data Type Returned
**	SQL_LONGVARBINARY	Varying-length binary string.	**
**	SQL_LONGVARCHAR	Varying-length Unicode character string.	**
NCHAR(<i>n</i>)	SQL_WCHAR	Fixed-length Unicode character string.	NCHAR(<i>n</i>)
NVARCHAR(<i>n</i>)	SQL_WVARCHAR	Varying-length Unicode character string.	NVARCHAR(<i>n</i>)
REAL	SQL_REAL	Signed, single precision, floating-point number.	REAL
SMALLINT	SQL_SMALLINT	Small signed, exact whole number.	SMALLINT
TIME(<i>p</i>)	SQL_TYPE_TIME	Time value.	TIME(<i>p</i>)
TIMESTAMP(<i>p</i>)	SQL_TYPE_TIMESTAMP	Date and time value.	TIMESTAMP(<i>p</i>)
TINYINT	SQL_TINYINT	Very small signed, exact whole number.	TINYINT
VARBINARY(<i>n</i>)	SQL_VARBINARY	Varying-length binary string.	VARBINARY(<i>n</i>)
VARCHAR(<i>n</i>)	SQL_VARCHAR	Varying-length character string.	VARCHAR(<i>n</i>)
**	SQL_WLONGVARCHAR	Varying-length Unicode character string.	**

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** The ODBC SQL data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

Data Types for Oracle

The following table lists the data type support for an Oracle database.

For data source specific information about Oracle data types, see the Oracle database documentation.

Table A2.12 Data Types for Oracle

Data Type Definition Keyword*	Oracle Data Type	Description	Data Type Returned
BIGINT	BIGINT	Large signed, exact whole number.	BIGINT
BINARY(<i>n</i>)	RAW(<i>n</i>)	Fixed or varying length binary string.	BINARY(<i>n</i>)
CHAR(<i>n</i>)	CHAR(<i>n</i>)	Fixed-length character string.	CHAR(<i>n</i>)
DATE	DATE	Date values.	TIMESTAMP(<i>p</i>)***
DECIMAL NUMERIC(<i>p,s</i>)	NUMBER(<i>p,s</i>)	Signed, fixed-point decimal number.	DOUBLE†
DOUBLE	BINARY_DOUBLE	Signed, double precision, floating-point number.	DOUBLE
FLOAT(<i>p</i>)	FLOAT(<i>p</i>)	Signed, single precision or double precision floating-point number.	FLOAT(<i>p</i>)
INTEGER	INTEGER	Regular signed, exact whole number.	INTEGER
**	LONG	Varying-length character string data.	**
NCHAR(<i>n</i>)	NCHAR(<i>n</i>)	Fixed-length Unicode character string.	NCHAR(<i>n</i>)
NVARCHAR(<i>n</i>)	NVARCHAR(<i>n</i>)	Varying-length Unicode character string.	NVARCHAR(<i>n</i>)
REAL	REAL	Signed, single precision floating-point number.	REAL
SMALLINT	SMALLINT	Small signed, exact whole number.	SMALLINT
TIME(<i>p</i>)	TIME(<i>p</i>)	Time value.	TIMESTAMP(<i>p</i>)***
TIMESTAMP(<i>p</i>)††	TIMESTAMP(<i>p</i>)	Date and time value.	TIMESTAMP(<i>p</i>)
TINYINT	TINYINT	Very small signed, exact whole number.	TINYINT
VARBINARY(<i>n</i>)	LONG RAW(<i>n</i>)	Varying-length binary string.	VARBINARY(<i>n</i>)

Data Type Definition Keyword*	Oracle Data Type	Description	Data Type Returned
VARCHAR(<i>n</i>)	VARCHAR2(<i>n</i>) ^{†††}	Varying-length character string.	VARCHAR(<i>n</i>)

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** The Oracle data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

*** The timestamp returned by the DATE and TIME data types can be changed to date and time values by using the DATEPART function with the PUT function.

† The ORNUMERIC= connection argument and table option determine how numbers read from or inserted into the Oracle NUMBER column are treated. ORNUMERIC=YES, which is the default, indicates that non-integer values with explicit precision are treated as NUMERIC values.

†† The TIMESTAMP(*p*) data type is not available on Z/OS.

††† The VARCHAR2(*n*) type is supported for up to 32,767 bytes if the Oracle version is 12c and the Oracle MAX_STRING_SIZE= parameter is set to EXTENDED.

Data Types for PostgreSQL

The following table lists the data type support for a PostgreSQL database.

The BINARY, NCHAR, NVARCHAR, TINYINT, and VARBINARY data types are not supported for data type definition.

For data source specific information about PostgreSQL data types, see the PostgreSQL database documentation.

Table A2.13 Data Types for PostgreSQL

Data Type Definition Keyword*	PostgreSQL Data Type	Description	Data Type Returned
BIGINT	BIGINT	Large signed, exact whole number. OR Signed eight-byte integer.	BIGINT
**	BIGSERIAL	Autoincrementing eight-byte integer.	**
**	BIT(<i>n</i>)	Fixed-length bit string.	**
**	BIT VARYING(<i>n</i>)	Variable-length bit string.	**
**	BOOLEAN	Logical Boolean (true/false).	**
**	BOX	Rectangular box on a plane.	**
**	BYTEA	Binary data (byte array).	**

Data Type Definition Keyword*	PostgreSQL Data Type	Description	Data Type Returned
CHAR(<i>n</i>)	CHAR(<i>n</i>)	Fixed-length character string.	CHAR(<i>n</i>)
**	CIDR	IPv4 or IPv6 network address.	**
**	CIRCLE	Circle on a plane.	**
DATE	DATE	Date value.	DATE
DECIMAL NUMERIC(<i>p,s</i>)	NUMERIC(<i>p,s</i>)	Signed, fixed-point decimal number.	DECIMAL NUMERIC(<i>p,s</i>)
DOUBLE	DOUBLE PRECISION	Signed, double precision, floating-point number.	DOUBLE
FLOAT(<i>p</i>)	FLOAT(<i>p</i>)	Signed, single precision or double precision, floating-point number.	FLOAT(<i>p</i>)
**	INET	IPv4 or IPv6 host address.	**
INTEGER	INTEGER	Regular signed, exact whole number.	INTEGER
**	INTERVAL	Time span.	**
**	LINE	Infinite line on a plane.	**
**	LSEG	Line segment on a plane.	**
**	MACADDR	Media Access Control address.	**
**	MONEY	Currency amount.	**
**	PATH	Geometric path on a plane.	**
**	POINT	Geometric point on a plane.	**
**	POLYGON	Closed geometric path on a plane.	**
REAL	REAL	Signed, single precision floating-point number.	REAL
**	SERIAL	Autoincrementing four-byte integer.	**

Data Type Definition Keyword*	PostgreSQL Data Type	Description	Data Type Returned
SMALLINT	SMALLINT	Small signed, exact whole number.	SMALLINT
**	SMALL SERIAL	Autoincrementing two-byte integer.	**
**	TEXT	Variable-length character string.	**
TIME(<i>p</i>)	TIME(<i>p</i>)	Time value.	TIME(<i>p</i>)
TIMESTAMP(<i>p</i>)	TIMESTAMP(<i>p</i>)	Date and time value.	TIMESTAMP(<i>p</i>)
**	TSQUERY	Text search query.	**
**	TSVECTOR	Text search document.	**
**	TXID_SNAPSHOT	User-level transaction ID snapshot.	**
**	UUID	Universally unique identifier.	**
VARCHAR(<i>n</i>)	CHARACTER VARYING(<i>n</i>)	Varying-length character string.	VARCHAR(<i>n</i>)
**	XML	XML data.	**

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** The PostgreSQL data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

Data Types for SAP

The following table lists the data type support for an SAP system.

For an SAP system, no data types are supported for column definition. Native ABAP SAP data types are mapped to similar data types for data retrieval only.

For data source specific information about the ABAP SAP data types, see the SAP system database documentation.

Table A2.14 Data Types for SAP

ABAP SAP Data Type	Description	Data Type Returned
ACCP	Posting period.	CHAR(<i>n</i>) for non-Unicode SAP system; NCHAR(<i>n</i>) for Unicode SAP system
CHAR	Fixed-length character string.	CHAR(<i>n</i>) for non-Unicode SAP system; NCHAR(<i>n</i>) for Unicode SAP system
CLNT	Client field.	CHAR(<i>n</i>) for non-Unicode SAP system; NCHAR(<i>n</i>) for Unicode SAP system
CUKY	Currency key. Fields of this type are referenced by fields of type CURR.	CHAR(<i>n</i>) for non-Unicode SAP system; NCHAR(<i>n</i>) for Unicode SAP system
CURR	Currency field. Corresponds to the DEC field. Field refers to a field of type CUKY.	CHAR(<i>n</i>)
DATS	Date values.	DATE
DEC	Signed, fixed-point decimal number.	CHAR(<i>n</i>)
FLTP	Floating-point number.	DOUBLE
INT1	Very small signed, exact whole number.	TINYINT
INT2	Small signed, exact whole number.	SMALLINT
INT4	Regular signed, exact whole number.	INTEGER
LANG	Language key, which has its own field format for special functions. The conversion exit ISOLA converts the value to be displayed to that of the database and the opposite is true.	CHAR(<i>n</i>) for non-Unicode SAP system; NCHAR(<i>n</i>) for Unicode SAP system
LCHR	Fixed-length character string.	VARCHAR(<i>n</i>) for non-Unicode SAP system; NVARCHAR(<i>n</i>) for Unicode SAP system
LRAW	Uninterpreted varying-length byte string.	VARBINARY(<i>n</i>)
NUMC	Text string.	CHAR(<i>n</i>) for non-Unicode SAP system; NCHAR(<i>n</i>) for Unicode SAP system
PREC	The precision of a QUAN field.	CHAR(<i>n</i>)
QUAN	A quantity that corresponds to the DEC field.	CHAR(<i>n</i>)
RAW	An uninterpreted byte string.	BINARY
TIMS	Time value.	TIME(<i>p</i>)

ABAP SAP Data Type	Description	Data Type Returned
UNIT	Units key and referenced by a QUAN data type.	CHAR(<i>n</i>) for non-Unicode SAP system; NCHAR(<i>n</i>) for Unicode SAP system
VARC	Varying-length character string data. As of SAP release 3.0, creating fields of this data type is no longer supported. Existing fields with this data type can be used, except in a WHERE condition in the SELECT statement.	VARCHAR(<i>n</i>) for non-Unicode SAS system; NVARCHAR(<i>n</i>) for Unicode SAP system

Data Types for SAP HANA

The following table lists the data type support for an SAP HANA database.

For data source specific information about the SAP HANA data types, see the SAP HANA database documentation.

Table A2.15 Data Types for SAP HANA

Data Type Definition Keyword *	SAP HANA Data Type	Description	Data Type Returned
**	ALPHANUM(<i>n</i>)	Varying-length character string.	NVARCHAR(<i>n</i>)
BIGINT	BIGINT	64-bit integer.	BIGINT
BINARY(<i>n</i>)	BINARY(<i>n</i>)	Fixed-length binary data.	BINARY(<i>n</i>)
**	BLOB	Varying-length binary large object string.	**
CHAR(<i>n</i>)	CHAR(<i>n</i>)	Varying-length character string.	CHAR(<i>n</i>)
**	CLOB	Varying-length character large object string.	**
DATE	DATE	Year, month, and day values.	DATE
DECIMAL NUMERIC(<i>p,s</i>)	DECIMAL(<i>p,s</i>)	Signed, exact, fixed-point decimal number.	DECIMAL(<i>p,s</i>)
DOUBLE	DOUBLE	Double-precision floating-point number.	DOUBLE
FLOAT(<i>p</i>)	FLOAT(<i>n</i>)	Floating-point number.	FLOAT(<i>p</i>)

Data Type Definition Keyword *	SAP HANA Data Type	Description	Data Type Returned
INTEGER	INTEGER	32-bit integer.	INTEGER
NCHAR(<i>n</i>)	NCHAR(<i>n</i>)	Fixed-length Unicode character string.	NCHAR(<i>n</i>)
**	NCLOB	Fixed-length character large object string.	**
NVARCHAR(<i>n</i>)	NVARCHAR(<i>n</i>)	Varying-length Unicode character large object string.	NVARCHAR(<i>n</i>)
REAL	REAL	Floating-point number.	REAL
**	SECONDATE	Date and time value.	**
**	SHORTTEXT(<i>n</i>)	Varying-length character string.	NVARCHAR(<i>n</i>)
**	SMALLDECIMAL(<i>p,s</i>)	Floating-point decimal number.	DECIMAL(<i>p,s</i>)
SMALLINT	SMALLINT	16-bit integer.	SMALLINT
**	TEXT	Varying-length Unicode character large object string.	**
TIME(<i>p</i>)	TIME(<i>p</i>)	Time value.	TIME(<i>p</i>)
TIMESTAMP(<i>p</i>)	TIMESTAMP(<i>p</i>)	Date and time value.	TIMESTAMP(<i>p</i>)
TINYINT	TINYINT	Unsigned 8-bit integer.	TINYINT
VARBINARY(<i>n</i>)	VARBINARY(<i>n</i>)	Varying-length binary string.	VARBINARY(<i>n</i>)
VARCHAR(<i>n</i>)	VARCHAR(<i>n</i>)	Varying-length character string.	VARCHAR(<i>n</i>)

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** The SAP HANA data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

Data Types for SASHDAT

The following table lists the data type support for a SASHDAT table. A SASHDAT table contains data that is added to the Hadoop Distributed File System (HDFS) by SAS.

The BIGINT, BINARY, DECIMAL, FLOAT, INTEGER, NCHAR, NVARCHAR, NUMERIC, REAL, SMALLINT, TINYINT, VARBINARY, and VARCHAR data types are not supported for data type definition.

A SASHDAT table is Write only for data and Read-Write for metadata information such as column attributes.

Table A2.16 Data Types for SASHDAT

Data Type Definition Keyword *	SASHDAT Data Type	Description
CHAR(<i>n</i>)	CHAR(<i>n</i>)	Fixed-length character string.
DATE	DATE	64-bit double precision, floating-point number. By default, applies the DATE9 SAS format.
DOUBLE	DOUBLE	64-bit double-precision, floating-point number.
TIME(<i>p</i>)	DOUBLE	64-bit double precision, floating-point number. By default, applies the TIME8 SAS format.
TIMESTAMP(<i>p</i>)	DOUBLE	64-bit double precision, floating-point number. By default, applies the DATETIME19.2 SAS format.

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

Data Types for Sybase IQ

The following table lists the data type support for a Sybase IQ database.

For data source specific information about the Sybase IQ database data types, see the Sybase IQ database documentation.

Table A2.17 Data Types for Sybase IQ

Data Type Definition Keyword*	Sybase IQ Data Type	Description	Data Type Returned
BIGINT	BIGINT	64-bit integer.	BIGINT
BINARY(<i>n</i>)	BINARY(<i>n</i>)	Fixed-length binary string.	BINARY(<i>n</i>)
***	BIT	Integer that stores only the values 0 or 1.	***

Data Type Definition Keyword*	Sybase IQ Data Type	Description	Data Type Returned
CHAR(<i>n</i>)	CHAR(<i>n</i>)	Varying-length character string.	VARCHAR(<i>n</i>)
DATE	DATE	Date values.	DATE
DECIMAL NUMERIC(<i>p,s</i>)	DECIMAL(<i>p,s</i>)	Signed, exact, fixed-point decimal number.	DECIMAL(<i>p,s</i>)
DOUBLE	DOUBLE	Double-precision floating-point number.	DOUBLE
FLOAT(<i>p</i>)	FLOAT(<i>p</i>)	Floating-point number.	FLOAT(<i>p</i>)
INTEGER	INTEGER	32-bit integer.	INTEGER
**	LONG BINARY	Varying-length binary string.	VARBINARY(<i>n</i>)
**	LONG VARBIT	Arbitrary length bit arrays.	VARCHAR(<i>n</i>)
NCHAR(<i>n</i>)	NCHAR(<i>n</i>)	Fixed-length Unicode character string.	NCHAR(<i>n</i>)
**	LONG NVARCHAR(<i>n</i>)	Varying length Unicode character string.	NVARCHAR(<i>n</i>)
**	MONEY	Fixed-point decimal number that stores monetary data.	DOUBLE
NVARCHAR(<i>n</i>)	NVARCHAR(<i>n</i>)	Varying-length Unicode character string.	NVARCHAR(<i>n</i>)
REAL	REAL	Floating-point number.	REAL
SMALLINT	SMALLINT	16-bit integer.	SMALLINT
TIME(<i>p</i>)	TIME(<i>p</i>)	Time value.	TIME(<i>p</i>)
TIMESTAMP(<i>p</i>)	TIMESTAMP(<i>p</i>)	Date and time value.	TIMESTAMP(<i>p</i>)
TINYINT	TINYINT	Unsigned 8-bit integer.	TINYINT
VARBINARY(<i>n</i>)	VARBINARY(<i>n</i>)	Varying length binary string.	VARBINARY(<i>n</i>)
VARCHAR(<i>n</i>)	CHAR(<i>n</i>)	Varying-length character string.	VARCHAR(<i>n</i>)

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE

can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** The Sybase IQ data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

*** The Sybase IQ data type cannot be defined or retrieved.

Data Types for Teradata

The following table lists the data type support for a Teradata database.

The NCHAR, NVARCHAR, and REAL data types are not supported for data type definition.

For data source specific information about the Teradata data types, see the Teradata database documentation.

Table A2.18 Data Types for Teradata

Data Type Definition Keyword*	Teradata Data Type	Description	Data Type Returned
BIGINT	BIGINT	Large signed, exact whole number.	BIGINT
BINARY(<i>n</i>)	BYTE(<i>n</i>)	Fixed-length binary string	BINARY(<i>n</i>)
**	BLOB	Large Binary Object.	**
CHAR(<i>n</i>)	CHAR(<i>n</i>)	Fixed-length character string.	CHAR(<i>n</i>)
**	CLOB	Large Character Object.	**
DATE	DATE	Date values.	DATE
DECIMAL NUMERIC(<i>p,s</i>)	DECIMAL(<i>p,s</i>)	Signed, fixed-point decimal number.	DECIMAL(<i>p,s</i>)
DOUBLE	FLOAT	Signed, double precision, floating-point number.	DOUBLE
FLOAT(<i>p</i>)	FLOAT(<i>p</i>)	Signed, double precision, floating-point number.	FLOAT(<i>p</i>)
INTEGER	INTEGER	Regular signed, exact whole number.	INTEGER
**	LONG VARCHAR	Varying-length character string.	**
SMALLINT	SMALLINT	Small signed, exact whole number	SMALLINT

Data Type Definition Keyword*	Teradata Data Type	Description	Data Type Returned
TIME(<i>p</i>)	TIME(<i>p</i>)	Time value.	TIME(<i>p</i>)
TIMESTAMP(<i>p</i>)	TIMESTAMP(<i>p</i>)	Date and time value.	TIMESTAMP(<i>p</i>)
TINYINT	BYTEINT	Very small signed, exact whole number.	TINYINT
VARBINARY(<i>n</i>)	VARBYTE(<i>n</i>)	Varying-length binary string.	VARBINARY(<i>n</i>)
VARCHAR(<i>n</i>)	VARCHAR(<i>n</i>)	Varying-length character string.	VARCHAR(<i>n</i>)

* The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

** The Teradata data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

Appendix 3

Using FedSQL and DS2

You can embed and execute FedSQL statements from within your DS2 programs. You can use FedSQL with DS2 in the following instances:

- You can invoke a DS2 package method expression as a function in a FedSQL SELECT statement.

For more information, see [“Using DS2 Packages in Expressions” on page 181](#).

- You can use the SQLSTMT package to generate, prepare, and execute FedSQL statements to update, insert, or delete rows from a table at run time.

The SQLSTMT package is intended for use with FedSQL statements that are executed multiple times, statements with parameters, or statements that generate a result set. For more information, see [“Using the SQLSTMT Package” in Chapter 16 of *SAS DS2 Language Reference*](#).

- You can also use the SQLEXEC function to generate, prepare, and execute FedSQL statements to update, insert, or delete rows from a table at run time.

The SQLEXEC function is intended for use with FedSQL statements that are executed only one time, do not have parameters, and do not produce a result set.

For more information, see the [“SQLEXEC Function” in *SAS DS2 Language Reference*](#).

- You can load data into a hash instance at run time by using a FedSQL SELECT statement in the DECLARE PACKAGE statement or the DATASET method.

For more information, see [“Using a FedSQL Query with a Hash Instance to Get Rows Dynamically at Run Time” in Chapter 16 of *SAS DS2 Language Reference*](#) the [“DATASET Method” in *SAS DS2 Language Reference*](#) and the [“DECLARE PACKAGE Statement, Hash Package” in *SAS DS2 Language Reference*](#).

- You can use the SET statement to input data by using a FedSQL SELECT statement.

For more information, see [“SET Statement with Embedded FedSQL” in Chapter 19 of *SAS DS2 Language Reference*](#) and the [“SET Statement” in *SAS DS2 Language Reference*](#).

Appendix 4

Tables Used in Examples

AfewWords	537
Customers	538
CustonLine	538
Densities	539
Depts	539
Employees	540
GrainProducts	540
Integers	541
Products	541
Sales	541
WorldCityCoords	542
WorldTemps	542

AfewWords

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

Table A4.1 *AfewWords*

Word1	Word2
*some/	WHERE
every	THING
no	BODY

Customers

Table A4.2 Customers

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

CustonLine

Table A4.3 CustonLine

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

Customer Number	BeginTime	EndTime
SP-N-44333958	01MAR2013:10:14:33.000	01MAR2013:10:35:27.908
GB-R-24994990	15MAR2013:09:00:06.000	15MAR2013:09:06:20.475

Densities

Table A4.4 Densities

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

Depts

DeptNo	DeptName	Manager
10	Sales	Jim Barnes
20	Research	Clifford James
30	Accounting	Barbara Sandman

DeptNo	DeptName	Manager
40	Operations	William Baylor

Employees

EmpID	Dept	Emp_Name	Pos	Hire_Date
1	10	Jim Barnes	Manager	2000-11-26
2	20	Clifford James	Manager	2000-11-26
3	30	Barbara Sandman	Manager	2000-11-26
4	40	William Baylor	Manager	2000-11-26
5	20	Greg Welty	Developer	2004-11-26
6	20	Penny Jackson	Developer	2004-11-26
7	10	Edward Murray	Sales Associate	2001-11-26
8	10	Ronald Thomas	Sales Associate	2002-11-26
9	30	Elsie Marks	Executive Assistant	2002-02-11
10	40	Bruno Kramer	Grounds support technician	2003-11-02

GrainProducts

Table A4.5 GrainProducts

Prodid	Product
1424	Rice
3421	Corn
3234	Wheat
3485	Oat

Integers

Table A4.6 Integers

BI (BIGINT data type)	II (INTEGER data type)	SI (SMALLINT data type)	TI (TINYINT data type)
9223372036854775800	2147483647	32767	127

Products

Table A4.7 Products

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

Sales

Table A4.8 Sales

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

WorldCityCoords

Table A4.9 *WorldCityCoords*

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

WorldTemps

Table A4.10 *WorldTemps*

City	Country	AvgHigh	AvgLow
Algiers	Algeria	90	45
Amsterdam	Netherlands	70	33
Beijing	China	86	17
Bombay	India	90	68
Calcutta	India	97	56

City	Country	AvgHigh	AvgLow
Caracas	Venezuela	83	57
Geneva	Switzerland	76	28
Hong Kong	China	89	51
Lagos	Nigeria	90	75
Madrid	Spain	89	36
Shanghai	China		33
Zurich	Switzerland	78	25

Appendix 5

DICTIONARY Table Descriptions

DICTIONARY.CATALOGS	545
DICTIONARY.COLUMNS	545
DICTIONARY.COLUMN STATISTICS	549
DICTIONARY.STATISTICS	550
DICTIONARY.TABLES	555

DICTIONARY.CATALOGS

The following table lists the columns that appear in the result table for the DICTIONARY.CATALOGS table.

Column Name	Column Number	Data Type	Description
CATALOG	1	NVARCHAR	Catalog name
DRIVER	2	NVARCHAR	Data source name
DESCRIPTION	3	NVARCHAR	Description of catalog

DICTIONARY.COLUMNS

The following table lists the columns that appear in the result table for the DICTIONARY.COLUMNS table.

Note: Null indicates either a null or a missing value.

Column Name	Column Number	Data Type	Description
TABLE_CAT	1	NVARCHAR*	Catalog name

Column Name	Column Number	Data Type	Description
TABLE_SCHEM	2	NVARCHAR	<p>Schema name</p> <p>Null is returned if the schema is not applicable to the data source. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have schemas.</p>
TABLE_NAME	3	NVARCHAR*	Table name
COLUMN_NAME	4	NVARCHAR*	<p>Column name</p> <p>The driver returns an empty string for a column that does not have a name.</p>
DATA_TYPE	5	signed INTEGER*	<p>TKTS data type.</p> <p>This can be a SAS FedSQL data type or a remote table driver-specific data type. For datetime and interval data types, this column returns the concise data type such as DATE or INTERVAL_YEAR_TO_MONTH, rather than the nonconcise data type such as DATETIME or INTERVAL.</p> <p>For information about driver-specific TKTS data types, see the remote table driver's documentation.</p>
TYPE_NAME	6	NVARCHAR*	Data source-dependent common data type name, for example, CHAR, NVARCHAR, or BIGINT.
COLUMN_SIZE	7	signed BIGINT*	<p>If DATA_TYPE is CHAR or VARCHAR, this column contains the maximum length of the column in characters. For datetime data types, this is the total number of characters required to display the value when it is converted to characters. For numeric data types, this is either the total number of digits or the total number of bits allowed in the column, according to the NUM_PREC_RADIX column. For interval data types, this is the number of characters in the character representation of the interval literal.</p>
BUFFER_LENGTH	8	signed BIGINT*	The length in bytes of data transferred. For numeric data, this size might be different from the size of the data stored on the data source. This value might be different from COLUMN_SIZE column for character data.

Column Name	Column Number	Data Type	Description
DECIMAL_DIGITS	9	signed INTEGER	<p>The total number of significant digits to the right of the decimal point.</p> <p>If DATA_TYPE is TIME and TIMESTAMP, this column contains the number of digits in the fractional seconds component. For the other data types, this is the decimal digits of the column on the data source. For interval data types that contain a time component, this column contains the number of digits to the right of the decimal point (fractional seconds). For interval data types that do not contain a time component, this column is 0. Null is returned for data types where DECIMAL_DIGITS is not applicable.</p>
NUM_PREC_RADIX	10	signed INTEGER	<p>For numeric data types, either 10 or 2 is returned.</p> <p>If the value returned is 10, the values in COLUMN_SIZE and DECIMAL_DIGITS contain the number of decimal digits allowed for the column. For example, a DECIMAL(12,5) column would return 10 for NUM_PREC_RADIX, 12 for COLUMN_SIZE, and 5 for DECIMAL_DIGITS; a FLOAT column could return 10 for NUM_PREC_RADIX, 15 for COLUMN_SIZE, and null for DECIMAL_DIGITS.</p> <p>If the value returned is 2, the values in COLUMN_SIZE and DECIMAL_DIGITS give the number of bits allowed in the column. For example, a FLOAT column could return 2 for RADIX, 53 for COLUMN_SIZE, and null for DECIMAL_DIGITS.</p> <p>Null is returned for data types where NUM_PREC_RADIX is not applicable.</p>
NULLABLE	11	signed INTEGER not null	<p>TKTS_NO_NULLS if the column will not accept null values.</p> <p>TKTS_NULLABLE if the column accepts null values.</p> <p>TKTS_NULLABLE_UNKNOWN if it is not known whether the column accepts null values.</p> <p>The value returned for this column is different from the value returned for the IS_NULLABLE column. The NULLABLE column indicates with certainty that a column can accept null values, but cannot indicate with certainty that a column does not accept null values. The IS_NULLABLE column indicates with certainty that a column cannot accept null values, but cannot indicate with certainty that a column accepts null values.</p>
REMARKS	12	NVARCHAR	<p>A description of the column. For a FedSQL view, the value in the REMARKS column is FedSQL.VIEW.</p>

Column Name	Column Number	Data Type	Description
COLUMN_DEF	13	NVARCHAR	<p>The default value of the column. The value in this column should be interpreted as a string if it is enclosed in quotation marks.</p> <p>If null was specified as the default value, then this column is the word NULL, not enclosed in quotation marks. If the default value cannot be represented without truncation, then this column contains TRUNCATED, not enclosed in quotation marks. If no default value is specified, then this column is null.</p> <p>The value of COLUMN_DEF can be used in generating a new column definition, except when it contains the value TRUNCATED.</p>
TKTS_DATA_TYPE	14	signed INTEGER*	<p>Data type, as it appears in the TKTS_DESC_TYPE record field.</p> <p>This can be a SAS FedSQL data type or a remote table driver-specific data type. This column is the same as the DATA_TYPE column, with the exception of datetime and interval data types. This column returns the nonconcise data type (such as DATETIME or INTERVAL), rather than the concise data type (such as DATE or YEAR_TO_MONTH) for datetime and interval data types. If this column returns DATETIME or INTERVAL, the specific data type can be determined from the TKTS_DATETIME_SUB column.</p> <p>For information about driver-specific TKTS data types, see the remote table driver's documentation.</p>
TKTS_DATETIME_SUB	15	signed INTEGER	<p>The subtype code for datetime and interval data types.</p> <p>For other data types, this column returns a null.</p>
CHAR_OCTET_LENGTH	16	signed BIGINT	<p>The maximum length in bytes of a character or binary data type column.</p> <p>For all other data types, this column returns a null.</p>
ORDINAL_POSITION	17	signed INTEGER*	<p>The ordinal position of the column in the table. The first column in the table is number 1.</p>
IS_NULLABLE	18	NVARCHAR*	<p>NO if the column does not include nulls.</p> <p>YES if the column could include nulls.</p> <p>This column returns a zero-length string if nullability is unknown.</p> <p>ISO rules are followed to determine nullability. An ISO SQL-compliant DBMS cannot return an empty string.</p> <p>The value returned for this column is different from the value returned for the NULLABLE column.</p>

* Value cannot be a null

DICTIONARY.COLUMN STATISTICS

The following table lists the columns that appear in the result table for the DICTIONARY.COLUMN_STATISTICS table.

Note: Null indicates either a null or a missing value.

Note: The first four columns do not appear in the result table.

Column Name	Column Number	Data Type	Description
TABLE_CAT	1	NVARCHAR*	Catalog name
TABLE_SCHEM	2	NVARCHAR	Schema name Null is returned if the schema is not applicable to the data source. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have schemas.
TABLE_NAME	3	NVARCHAR*	Table name
COLUMN_NAME	4	NVARCHAR*	Column name The driver returns an empty string for a column that does not have a name.
TYPE	5	unsigned INTEGER*	Type of information being returned: TKTS_COL_STAT indicates a statistic for the columns specified in the CARDINALITY and HISTOGRAM columns.
CARDINALITY	6	unsigned INTEGER	Cardinality of column or column set in table. This is the number of distinct, non-null values in the column or column set. Null is returned if the value is not available from the data source.
NULL_FRAC	7	DOUBLE	Fraction (expressed as a decimal) of the column's entries that are null. If any of the columns in a set of columns is null then that entry is considered null.
MOST_COMMON_VALS	8	NVARCHAR	Contains a value that is common for this column. Null if there are no common values. Null if the number of columns is greater than one.

Column Name	Column Number	Data Type	Description
MOST_COMMON_FREQS	9	DOUBLE	<p>A frequency of the common value returned in MOST_COMMON_VALS.</p> <p>This would likely be the number of occurrences of that particular column value divided by the total number of rows in the table.</p> <p>Null if the number of columns is greater than one.</p>
HISTOGRAM_ENTRY	10	unsigned INTEGER	<p>The entry for the HISTOGRAM_BOUNDS value.</p> <p>Null if the number of columns is greater than one.</p>
HISTOGRAM_BOUNDS	11	NVARCHAR	<p>One entry in a list of values that divide the column's values into groups of approximately equal population.</p> <p>The most common values (returned by MOST_COMMON_VALS) are omitted from the histogram calculation. No HISTOGRAM_BOUNDS are returned if the column's data type does not have a less than (<) operator, or if the values returned by MOST_COMMON_VALS account for the entire population of the column.</p> <p>Null if the number of columns is greater than one.</p>
CORRELATION	12	DOUBLE	<p>Statistical correlation between the physical row ordering and the logical ordering of the column values.</p> <p>The values range from -1 to +1. This value is not returned if the column's data type does not have a less than (<) operator.</p> <p>Null if the number of columns is greater than one.</p>

* Value cannot be a null or missing value

DICTIONARY.STATISTICS

The following table lists the columns that appear in the result table for the DICTIONARY.STATISTICS table.

Note: Null indicates either a null or a missing value.

Column Name	Column Number	Data Type	Description
TABLE_CAT	1	NVARCHAR*	TKTS catalog name.

Column Name	Column Number	Data Type	Description
TABLE_SCHEM	2	NVARCHAR	<p>Schema name of the table to which the statistic or index applies.</p> <p>Null if not applicable to the data source. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have schemas.</p>
TABLE_NAME	3	NVARCHAR not null	Name of the table to which the statistic or index applies.
NON_UNIQUE	4	signed INTEGER	<p>Indicates whether the index prohibits duplicate values: TKTS_TRUE if the index values can be non-unique. TKTS_FALSE if the index values must be unique.</p> <p>Null is returned if TYPE is TKTS_TABLE_STAT.</p>
INDEX_QUALIFIER	5	NVARCHAR	<p>The identifier that is used to qualify the index name doing a DROP INDEX.</p> <p>Null is returned if an index qualifier is not supported by the data source or if TYPE is TKTS_TABLE_STAT. If a non-null value is returned in this column, it must be used to qualify the index name on a DROP INDEX statement; otherwise, the TABLE_SCHEM should be used to qualify the index name.</p>
INDEX_NAME	6	NVARCHAR	<p>Index name.</p> <p>Null is returned if TYPE is TKTS_TABLE_STAT.</p>
TYPE	7	signed INTEGER*	<p>Type of information being returned:</p> <p>TKTS_TABLE_STAT indicates a statistic for the table in the CARDINALITY or PAGES column.</p> <p>TKTS_INDEX_BTREE indicates a B-Tree index.</p> <p>TKTS_INDEX_CLUSTERED indicates a clustered index.</p> <p>TKTS_INDEX_CONTENT indicates a content index.</p> <p>TKTS_INDEX_HASHED indicates a hashed index.</p> <p>TKTS_INDEX_OTHER indicates another type of index.</p>
ORDINAL_POSITION	8	signed INTEGER	<p>Column sequence number in index starting with 1.</p> <p>Null is returned if TYPE is TKTS_TABLE_STAT.</p>
COLUMN_NAME	9	NVARCHAR	<p>Column name.</p> <p>If the column is based on an expression, such as SALARY + BENEFITS, the expression is returned; if the expression cannot be determined, an empty string is returned. Null is returned if TYPE is TKTS_TABLE_STAT.</p>

Column Name	Column Number	Data Type	Description
ASC_OR_DESC	10	CHAR(1)	Sort sequence for the column: A for ascending; D for descending. Null is returned if column sort sequence is not supported by the data source or if TYPE is TKTS_TABLE_STAT.
CARDINALITY	11	signed BIGINT	Cardinality of table or index. Number of rows in table if TYPE is TKTS_TABLE_STAT. Number of unique values in the index if TYPE is not TKTS_TABLE_STAT. Null if the value is not available from the data source.
PAGES	12	signed INTEGER	Number of pages used to store the index or table. Number of pages for the table if TYPE is TKTS_TABLE_STAT. Number of pages for the index if TYPE is not TKTS_TABLE_STAT. Null if the value is not available from the data source or if not applicable to the data source.
FILTER_CONDITION	13	NVARCHAR	If the index is a filtered index, this is the filter condition, such as SALARY > 30000. If the filter condition cannot be determined, the value is an empty string. Null if the index is not a filtered index, if it cannot be determined whether the index is a filtered index, or if TYPE is TKTS_TABLE_STAT.
AVG_FANOUT**	14	DOUBLE	The average fan-out of internal nodes for an index. Valid for TKTS_INDEX_* TYPE records. Null if not known or available.
MAX_FANOUT**	15	unsigned INTEGER	The maximum fan-out of internal nodes for an index. Valid for TKTS_INDEX_* TYPE records. Null if not known or available.
MIN_FANOUT**	16	unsigned INTEGER	The minimum fan-out of internal nodes for an index. Valid for TKTS_INDEX_* TYPE records. Null if not known or available.
INDEX_LEVELS**	17	unsigned INTEGER	The number of levels in an index. Valid for TKTS_INDEX_* TYPE records. Null if not known or available.
LEAF_LEVEL_BLOCKS**	18	unsigned INTEGER	The number of blocks at the leaf level of an index. Valid for TKTS_INDEX_* TYPE records. Null if not known or available.

Column Name	Column Number	Data Type	Description
LOCAL**	19	NVARCHAR	<p>Whether a table is local or must be accessed over a network. Valid for TKTS_TABLE_STAT TYPE records.</p> <p>Values YES, NO or Null if unknown.</p>
PARTITION_SCHEME**	20	unsigned INTEGER	<p>Whether the table is partitioned and, if so, does it use a round-robin, hash- or range-partitioning scheme. Valid for TKTS_TABLE_STAT TYPE records.</p> <p>TKTS_PARTITION_NONE indicates the table is not partitioned.</p> <p>TKTS_PARTITION_ROUND_ROBIN indicates a round-robin partitioning scheme.</p> <p>TKTS_PARTITION_HASH indicates a hash partitioning scheme.</p> <p>TKTS_PARTITION_RANGE indicates a range partitioning scheme. Null if not known or available.</p>
FRAGMENTATION**	21	unsigned INTEGER	<p>If the data is distributed, indicates whether it uses vertical fragmentation, horizontal fragmentation, derived fragmentation, or a hybrid fragmentation. Valid for TKTS_TABLE_STAT TYPE records.</p> <p>TKTS_FRAG_NONE indicates the data is not distributed. TKTS_FRAG_VERTICAL indicates vertical fragmentation.</p> <p>TKTS_FRAG_HORIZONTAL indicates horizontal fragmentation. TKTS_FRAG_DERIVED indicates derived fragmentation. TKTS_FRAG_HYBRID uses hybrid fragmentation. Null if not known or available.</p>
LABEL**	22	NVARCHAR (max 256 characters)	<p>Label associated with the table. Valid for TKTS_TABLE_STAT TYPE records.</p> <p>Null if not known or available.</p>
APPLICATION_TYPE**	23	NVARCHAR (max 8 characters)	<p>SAS application-specific type associated with the table. The value of this type field is created, stored, and retrieved by some SAS applications. The meaning of the value is specific to the application that created it. Valid for TKTS_TABLE_STAT TYPE records.</p> <p>Null if not known or available.</p>

Column Name	Column Number	Data Type	Description
PROTECTED**	24	signed INTEGER	<p>Indicates whether table is password protected. Valid for TKTS_TABLE_STAT TYPE records.</p> <p>TKTS_PROTECTION_NONE indicates table is not password protected.</p> <p>TKTS_PROTECTION_READ indicates table is Read protected.</p> <p>TKTS_PROTECTION_WRITE indicates table is Write protected.</p> <p>TKTS_PROTECTION_ALTER indicates table is alter protected.</p> <p>TKTS_PROTECTION_ENCRYPTED indicates table is encrypted.</p> <p>Null if not known or available.</p>
COMPRESS**	25	NVARCHAR (max 8 characters)	<p>Indicates whether the records in the table are compressed. Valid for TKTS_TABLE_STAT TYPE records.</p> <p>NO Indicates records are uncompressed.</p> <p>YES CHAR indicates RLE(Run Length Encoding) used to compress records.</p> <p>BINARY indicates RDC(Ross Data Compression) used to compress records.</p> <p>Null if not known or available.</p>
CHAR_CEI**	26	unsigned INTEGER	<p>Encoding value used for the table. Valid for TKTS_TABLE_STAT TYPE records.</p> <p>Null if not known or available.</p>
DELETED_ROWS**	27	signed BIGINT	<p>Number of deleted rows in the table. Valid for TKTS_TABLE_STAT TYPE records.</p> <p>Null if not known or available.</p>
DATE_CREATED**	28	DOUBLE	<p>Date the table was created. Value is a DATETIME that is defined by SAS. Valid for TKTS_TABLE_STAT TYPE records.</p> <p>Null if not known or available.</p>
DATE_MODIFIED**	29	DOUBLE	<p>Date the table was last modified. Value is a DATETIME that is defined by SAS. Valid for TKTS_TABLE_STAT TYPE records.</p> <p>Null if not known or available.</p>
TABLE_ATTRIBUTES**	30	signed INTEGER	<p>Attributes that apply to the table. This column represents a set of flags to convey specific attributes. A value of 0 for any flag indicates either a False or unknown condition. Valid for TKTS_TABLE_STAT TYPE records.</p> <p>Null if not known or available.</p>

Column Name	Column Number	Data Type	Description
ROW_LENGTH**	31	signed BIGINT	Length of the rows in the table on disk. Valid for TKTS_TABLE_STAT TYPE records. Null if not known or available.

* Value cannot be a null or missing value

** Optional-not defined in ODBC

DICTIONARY.TABLES

The following table lists the columns that appear in the result table for the DICTIONARY.TABLES table.

Note: Null indicates either a null or a missing value.

Column name	Column number	Data type	Description
TABLE_CAT	1	NVARCHAR not Null	Catalog name
TABLE_SCHEM	2	NVARCHAR	Schema name; null if not applicable to the data source. If a driver supports schemas for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have schemas.
TABLE_NAME	3	NVARCHAR not null	Table name
TABLE_TYPE	4	NVARCHAR not null	Table type name; one of the following: TABLE, VIEW, SYSTEM TABLE, GLOBAL TEMPORARY, LOCAL TEMPORARY, ALIAS, SYNONYM, or a data source-specific type name. The meanings of ALIAS and SYNONYM are driver-specific.
REMARKS	5	NVARCHAR	A description of the table
NATIVE_CAT	6	NVARCHAR	The native data store's Catalog; null if not applicable to the data source. If a driver supports catalogs for some tables but not for others, such as when the driver retrieves data from different DBMSs, it returns an empty string ("") for those tables that do not have catalogs.

Appendix 6

Usage Notes

Teradata: Enhancing Performance of FedSQL on Teradata

FedSQL queries that access system catalog tables (for example, the UPDATE statement, DROP TABLE statement, etc.) depend on joins of the system tables to obtain a result set from the Teradata database. In environments that create and drop a large number of database objects, performance suffers if the statistics about these system tables are not current. The following commands can be added to regular database maintenance processes to keep system table statistics up-to-date.

```
drop stats on dbc.tvn;
drop stats on dbc.owners;
drop stats on dbc.dbase;
drop stats on dbc.accessrights;
drop stats on dbc.tvfields;

collect stats on dbc.tvn column (tvnId);
collect stats on dbc.tvn INDEX ( DatabaseId ,TVNNameI );
collect stats on dbc.tvn column (DatabaseId );
collect stats on dbc.owners INDEX (ownerId);
collect stats on dbc.dbase INDEX ( DatabaseId );
collect stats on dbc.dbase column(JournalId);
collect stats on dbc.accessrights INDEX ( UserId ,DatabaseId );
collect stats on dbc.accessrights INDEX ( TVMId );
collect stats on dbc.accessrights column ( UserId ,TVMId);
collect stats on dbc.accessrights column (DatabaseId);
collect stats on dbc.tvfields column (DatabaseId);
collect stats on dbc.tvfields column (FieldId);
collect stats on dbc.tvfields column (tableId);

/* help stats displays what's there, this is optional */

help stats dbc.tvn;
help stats dbc.owners;
help stats dbc.dbase;
help stats dbc.accessrights;
help stats dbc.tvfields;

/* routine recollect, if stats have not been dropped */

collect stats on dbc.tvn ;
```

```
collect stats on dbc.owners ;  
collect stats on dbc.dbase ;  
collect stats on dbc.accessrights ;  
  
collect stats on dbc.tvfields ;
```

Appendix 7

ICU License

ICU License - ICU 1.8.1 and later	559
Third-Party Software Licenses	560
1. Unicode Data Files and Software	560
2. Chinese/Japanese Word Break Dictionary Data (cjdict.txt)	561
Lao Word Break Dictionary Data (laodict.txt)	564
3. Time Zone Database	565

ICU License - ICU 1.8.1 and later

COPYRIGHT AND PERMISSION NOTICE

Copyright (c) 1995-2014 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.

Third-Party Software Licenses

This section contains third-party software notices and/or additional terms for licensed third-party software components included within ICU libraries.

1. Unicode Data Files and Software

EXHIBIT 1

UNICODE, INC. LICENSE AGREEMENT - DATA FILES AND SOFTWARE

Unicode Data Files include all data files under the directories <http://www.unicode.org/Public/>, <http://www.unicode.org/reports/>, and <http://www.unicode.org/cldr/data/>. Unicode Data Files do not include PDF online code charts under the directory <http://www.unicode.org/Public/>. Software includes any source code published in the Unicode Standard or under the directories <http://www.unicode.org/Public/>, <http://www.unicode.org/reports/>, and <http://www.unicode.org/cldr/data/>.

NOTICE TO USER: Carefully read the following legal agreement. BY DOWNLOADING, INSTALLING, COPYING OR OTHERWISE USING UNICODE INC.'S DATA FILES ("DATA FILES"), AND/OR SOFTWARE ("SOFTWARE"), YOU UNEQUIVOCALLY ACCEPT, AND AGREE TO BE BOUND BY, ALL OF THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU DO NOT AGREE, DO NOT DOWNLOAD, INSTALL, COPY, DISTRIBUTE OR USE THE DATA FILES OR SOFTWARE.

COPYRIGHT AND PERMISSION NOTICE

Copyright © 1991-2014 Unicode, Inc. All rights reserved. Distributed under the Terms of Use in <http://www.unicode.org/copyright.html>.

Permission is hereby granted, free of charge, to any person obtaining a copy of the Unicode data files and any associated documentation (the "Data Files") or Unicode software and any associated documentation (the "Software") to deal in the Data Files or Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Data Files or Software, and to permit persons to whom the Data Files or Software are furnished to do so, provided that (a) the above copyright notice(s) and this permission notice appear with all copies of the Data Files or Software, (b) both the above copyright notice(s) and this permission notice appear in associated documentation, and (c) there is clear notice in each modified Data File or in the Software as well as in the documentation associated with the Data File(s) or Software that the data or software has been modified.

THE DATA FILES AND SOFTWARE ARE PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS

INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THE DATA FILES OR SOFTWARE.

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

Unicode and the Unicode logo are trademarks of Unicode, Inc. in the United States and other countries. All third party trademarks referenced herein are the property of their respective owners.

2. Chinese/Japanese Word Break Dictionary Data (cjdict.txt)

```
# The Google Chrome software developed by Google is licensed under
# the BSD license. Other software included in this distribution is provided
# under other licenses, as set forth below.
#
# The BSD License
# http://opensource.org/licenses/bsd-license.php
# Copyright (C) 2006-2008, Google Inc.
#
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions are met:
#
# Redistributions of source code must retain the above copyright notice, this
# list of conditions and the following disclaimer.
#
# Redistributions in binary form must reproduce the above copyright notice,
# this list of conditions and the following disclaimer in the documentation
# and/or other materials provided with the distribution.
#
# Neither the name of Google Inc. nor the names of its contributors may be
# used to endorse or promote products derived from this software without
# specific prior written permission.
#
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
# AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
# IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
# ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
# LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
# CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
# SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
# INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
# CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
# ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
# THE POSSIBILITY OF SUCH DAMAGE.
#
#
# The word list in cjdict.txt are generated by combining three word lists
# listed below with further processing for compound word breaking. The
```

```

# frequency is generated with an iterative training against Google
# web corpora.
#
# * Libtabe (Chinese)
#   - https://sourceforge.net/project/?group\_id=1519
#   - Its license terms and conditions are shown below.
#
# * IPADIC (Japanese)
#   - http://chasen.aist-nara.ac.jp/chasen/distribution.html
#   - Its license terms and conditions are shown below.
#
# -----COPYING.libtabe ---- BEGIN-----
#
# /*
#  * Copyright (c) 1999 TaBE Project.
#  * Copyright (c) 1999 Pai-Hsiang Hsiao.
#  * All rights reserved.
#  *
#  * Redistribution and use in source and binary forms, with or without
#  * modification, are permitted provided that the following conditions
#  * are met:
#  *
#  * . Redistributions of source code must retain the above copyright
#  *   notice, this list of conditions and the following disclaimer.
#  * . Redistributions in binary form must reproduce the above copyright
#  *   notice, this list of conditions and the following disclaimer in
#  *   the documentation and/or other materials provided with the
#  *   distribution.
#  * . Neither the name of the TaBE Project nor the names of its
#  *   contributors may be used to endorse or promote products derived
#  *   from this software without specific prior written permission.
#  *
#  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
#  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
#  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
#  * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
#  * REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
#  * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
#  * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
#  * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
#  * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
#  * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
#  * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
#  * OF THE POSSIBILITY OF SUCH DAMAGE.
#  */
#
# /*
#  * Copyright (c) 1999 Computer Systems and Communication Lab,
#  *                               Institute of Information Science, Academia Sinica.
#  * All rights reserved.
#  *
#  * Redistribution and use in source and binary forms, with or without
#  * modification, are permitted provided that the following conditions
#  * are met:
#  *
#  * . Redistributions of source code must retain the above copyright

```

```

# * notice, this list of conditions and the following disclaimer.
# * . Redistributions in binary form must reproduce the above copyright
# * notice, this list of conditions and the following disclaimer in
# * the documentation and/or other materials provided with the
# * distribution.
# * . Neither the name of the Computer Systems and Communication Lab
# * nor the names of its contributors may be used to endorse or
# * promote products derived from this software without specific
# * prior written permission.
# *
# * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
# * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
# * REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
# * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
# * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
# * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
# * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
# * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
# * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
# * OF THE POSSIBILITY OF SUCH DAMAGE.
# */
#
# Copyright 1996 Chih-Hao Tsai @ Beckman Institute, University of Illinois
# c-tsai4@uiuc.edu http://casper.beckman.uiuc.edu/~c-tsai4
#
# -----COPYING.libtabe-----END-----
#
# -----COPYING.ipadic-----BEGIN-----
#
# Copyright 2000, 2001, 2002, 2003 Nara Institute of Science
# and Technology. All Rights Reserved.
#
# Use, reproduction, and distribution of this software is permitted.
# Any copy of this software, whether in its original form or modified,
# must include both the above copyright notice and the following
# paragraphs.
#
# Nara Institute of Science and Technology (NAIST),
# the copyright holders, disclaims all warranties with regard to this
# software, including all implied warranties of merchantability and
# fitness, in no event shall NAIST be liable for
# any special, indirect or consequential damages or any damages
# whatsoever resulting from loss of use, data or profits, whether in an
# action of contract, negligence or other tortuous action, arising out
# of or in connection with the use or performance of this software.
#
# A large portion of the dictionary entries
# originate from ICOT Free Software. The following conditions for ICOT
# Free Software applies to the current dictionary as well.
#
# Each User may also freely distribute the Program, whether in its
# original form or modified, to any third party or parties, PROVIDED
# that the provisions of Section 3 ("NO WARRANTY") will ALWAYS appear

```

```

# on, or be attached to, the Program, which is distributed substantially
# in the same form as set out herein and that such intended
# distribution, if actually made, will neither violate or otherwise
# contravene any of the laws and regulations of the countries having
# jurisdiction over the User or the intended distribution itself.
#
# NO WARRANTY
#
# The program was produced on an experimental basis in the course of the
# research and development conducted during the project and is provided
# to users as so produced on an experimental basis. Accordingly, the
# program is provided without any warranty whatsoever, whether express,
# implied, statutory or otherwise. The term "warranty" used herein
# includes, but is not limited to, any warranty of the quality,
# performance, merchantability and fitness for a particular purpose of
# the program and the nonexistence of any infringement or violation of
# any right of any third party.
#
# Each user of the program will agree and understand, and be deemed to
# have agreed and understood, that there is no warranty whatsoever for
# the program and, accordingly, the entire risk arising from or
# otherwise connected with the program is assumed by the user.
#
# Therefore, neither ICOT, the copyright holder, or any other
# organization that participated in or was otherwise related to the
# development of the program and their respective officials, directors,
# officers and other employees shall be held liable for any and all
# damages, including, without limitation, general, special, incidental
# and consequential damages, arising out of or otherwise in connection
# with the use or inability to use the program or any product, material
# or result produced or otherwise obtained by using the program,
# regardless of whether they have been advised of, or otherwise had
# knowledge of, the possibility of such damages at any time during the
# project or thereafter. Each user will be deemed to have agreed to the
# foregoing by his or her commencement of use of the program. The term
# "use" as used herein includes, but is not limited to, the use,
# modification, copying and distribution of the program and the
# production of secondary products from the program.
#
# In the case where the program, whether in its original form or
# modified, was distributed or delivered to or received by a user from
# any person, organization or entity other than ICOT, unless it makes or
# grants independently of ICOT any specific warranty to the user in
# writing, such person, organization or entity, will also be exempted
# from and not be held liable to the user for any such damages as noted
# above as far as the program is concerned.
#
# -----COPYING.ipadic-----END-----

```

Lao Word Break Dictionary Data (laodict.txt)

```

# Copyright (c) 2013 International Business Machines Corporation
# and others. All Rights Reserved.
#
# Project:      http://code.google.com/p/lao-dictionary/

```

```

# Dictionary: http://lao-dictionary.googlecode.com/git/Lao-Dictionary.txt
# License:    http://lao-dictionary.googlecode.com/git/Lao-Dictionary-LICENSE.txt
#             (copied below)
#
# This file is derived from the above dictionary, with slight modifications.
# -----
# Copyright (C) 2013 Brian Eugene Wilson, Robert Martin Campbell.
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without modification,
# are permitted provided that the following conditions are met:
#
#     Redistributions of source code must retain the above copyright notice, this
#     list of conditions and the following disclaimer. Redistributions in binary
#     form must reproduce the above copyright notice, this list of conditions and
#     the following disclaimer in the documentation and/or other materials
#     provided with the distribution.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
# ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
# WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
# DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
# ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
# (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
# LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
# ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
# (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
# SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
# -----

```

3. Time Zone Database

ICU uses the public domain data and code derived from [Time Zone Database](#) for its time zone support. The ownership of the TZ database is explained in [BCP 175: Procedure for Maintaining the Time Zone Database](#) section 7.

7. Database Ownership

The TZ database itself is not an IETF Contribution or an IETF document. Rather it is a pre-existing and regularly updated work that is in the public domain, and is intended to remain in the public domain. Therefore, BCPs 78 [RFC5378] and 79 [RFC3979] do not apply to the TZ Database or contributions that individuals make to it. Should any claims be made and substantiated against the TZ Database, the organization that is providing the IANA Considerations defined in this RFC, under the memorandum of understanding with the IETF, currently ICANN, may act in accordance with all competent court orders. No ownership claims will be made by ICANN or the IETF Trust on the database or the code. Any person making a contribution to the database or code waives all rights to future claims in that contribution or in the TZ Database.

Index

Special Characters

\$BASE64Xw. format [81](#)
 \$BINARYw. format [82](#)
 \$CHARw. format [83](#), [89](#)
 \$Fw. format [89](#)
 \$HEXw. format [83](#)
 \$OCTALw. format [84](#)
 \$QUOTEw. format [85](#)
 \$REVERJw. format [87](#)
 \$REVERSw. format [88](#)
 \$UPCASEw. format [89](#)
 \$w. format [89](#)

A

ABS function [190](#)
 ACOS function [191](#)
 aggregate functions [179](#), [182](#)
 ANSI Standard for SQL and [504](#)
 calling Base SAS functions instead of
 [183](#)
 syntax [180](#)
 ALTER TABLE statement [351](#)
 ALTER= table option [426](#)
 ANSI Standard for SQL [503](#)
 arithmetic expressions
 type conversion for [25](#)
 ASIN function [192](#)
 ASYNCINDEX= table option [427](#)
 ATAN function [193](#)
 ATAN2 function [194](#)
 AVG function [196](#)

B

BAND function [197](#)
 BEGIN statement [355](#)
 BESTDw.p format [92](#)
 BESTw. format [90](#)
 BETA function [198](#)
 BETWEEN predicate [322](#)

BINARYw. format [93](#)
 binomial distributions
 probabilities from [282](#)
 bivariate normal distribution
 probability computed from [283](#)
 BL_ALLOW_READ_ACCESS= table
 option [428](#)
 BL_COPY_LOCATION= table option
 [428](#)
 BL_CPU_PARALLELISM= table option
 [429](#)
 BL_DATA_BUFFER_SIZE= table option
 [430](#)
 BL_DEFAULT_DIR= table option [431](#)
 BL_DISK_PARALLELISM= table option
 [431](#)
 BL_ERRORS= table option [432](#)
 BL_EXCEPTION= table option [433](#)
 BL_INDEXING_MODE= table option
 [434](#)
 BL_LOAD_REPLACE= table option [435](#)
 BL_LOAD= table option [434](#)
 BL_LOG= table option [435](#)
 BL_LOGFILE= table option [436](#)
 BL_OPTIONS= table option [437](#)
 BL_PARALLEL= table option [438](#)
 BL_PORT_MAX= table option [438](#)
 BL_PORT_MIN= table option [439](#)
 BL_RECOVERABLE= table option [439](#)
 BL_REMOTE_FILE= table option [440](#)
 BL_SKIP_INDEX_MAINTENANCE=
 table option [441](#)
 BL_SKIP_UNUSABLE_INDEXES=
 table option [442](#)
 BL_SKIP= table option [441](#)
 BL_WARNING_COUNT= table option
 [443](#)
 Black model
 call prices for European options on
 futures [199](#)

- put prices for European options on futures 201
- Black-Scholes model
 - call prices for European options on stocks 203
- BLACKCLPRC function 199
- BLACKPTPRC function 201
- BLKSHCLPRC function 203
- BLKSHPTPRC function 205
- BLOCKSIZE= table option 443
- BLSHIFT function 207
- BOR function 208
- BRSHIFT function 209
- BUFNO= table option 444
- BUFSIZE table option 445
- BULKLOAD= table option 446
- BULKOPTS= table option 447
- BXOR function 210

C

- call prices
 - European options on futures, Black model 199
 - European options on stocks, Black-Scholes model 203
 - for European options, based on Margrabe model 263
- CASE expression 322
- CAST function 210
- CEIL function 212
- CEILZ function 213
- character set conversions
 - error reporting 26
- CHARACTER_LENGTH function 215
- chi-squared distributions
 - probabilities 284
- COALESCE expression 326
- COALESCE function 216
- COALESCEC function 217
- column modifiers 503
- COMMAw.d format 94
- COMMAXw.d format 95
- COMMIT statement 356
- COMPRESS= table option 448
- concatenation expressions
 - type conversion for 26
- CONNECTION TO component of FROM clause 59
- constants
 - date, time, and datetime 52
- COPIES= table option 450
- correlated subqueries 45
- COS function 218
- COSH function 219
- COT function 220

- COUNT function 221
- CREATE INDEX statement 357
- CREATE TABLE statement 359
- CREATE VIEW statement 375
- CSS function 222
- CURRENT_DATE function 223
- CURRENT_LOCALE function 224
- CURRENT_TIME function 225
- CURRENT_TIME_GMT function 225
- CURRENT_TIMESTAMP function 226
- CURRENT_TIMESTAMP_GMT function 227
- CV function 228

D

- data sets
 - date, time, and datetime formats for 54
 - dates and times in 52
 - identifiers 505
 - naming conventions 505
- data source connection 12
- data sources 11
- data type conversions
 - error reporting 26
- data types 13
 - type conversion 23
- DATE function 229
- DATE functions 228
- DATEAMPW.d format 99
- DATEJUL function 229
- DATEPART function 230
- dates and times 51
 - date, time, and datetime constants 52
 - date, time, and datetime formats 54
 - date, time, and datetime functions 54
 - in SAS data sets and SPD Engine data sets 52
 - overview 51
- datetimes
 - See *dates and times*
- DATETIMEw.d format 101
- DATEw. format 98
- DAY function 231
- DAYw. format 103
- DBCREATE_INDEX_OPTS= table option 450
- DBCREATE_TABLE_OPTS= table option 451
- DDMMYYw. format 104
- DDMMYYxw. format 105
- DEGREES function 232
- DELETE statement 377
- delimited identifiers 17, 18
- DESCRIBE VIEW statement 376
- DICTIONARY tables 58

ANSI Standard for SQL and 505
 DISTINCT predicate 327
 DOLLARw.d format 107
 DOLLARXw.d format 108
 DOWNAMEw. format 109
 DROP INDEX statement 378
 DROP TABLE statement 379
 DROP VIEW statement 380
 DS2 and FedSQL 11
 DS2 package expressions
 as functions 181, 504
 DTDATEw. format 110
 DTMONYYw. format 111
 DTWKDATXw. format 113
 DTYEARw. format 114
 DTYQCw. format 115
 Dw.p format 96

E

E function 233
 ENCRYPT= table option 453
 ENCRYPTKEY table option 455
 ENDOBS= table option 458
 European options on futures
 call prices, based on Black model 199
 put prices, based on Black model 201
 European options on stocks
 call prices, based on Black-Scholes
 model 203
 call prices, based on Margrabe model
 263
 put prices based on Margrabe model
 265
 EUROW.d format 117
 EUROXw.d format 118
 Ew. format 116
 EXECUTE statement 381
 FedSQL Pass-Through Facility 59
 EXISTS predicate 328
 EXP function 234
 expressions 42, 321, 339
 arithmetic 25
 complex 341
 concatenation 26
 correlated subqueries 45
 DS2 package expressions 181, 504
 functions in 341
 logical 24
 order of evaluation 341
 query expressions 42
 relational 25
 simple 341
 subqueries 44, 341
 type conversion 23
 unary 24

value expressions 50

EXTENDOBSCOUNTER= table option
 458

F

FCMP procedure functions 504
 federated query
 definition and example 12
 Federated Query Language SQL
 (FedSQL)
 limitations and ANSI Standard for SQL
 505
 Federated SQL (FedSQL) 10
 FedSQL language
 benefits 11
 data source connection 12
 data sources 11
 description 10
 submitting programs 10
 FedSQL pass-through facility 59
 FLOATw.d format 119
 FLOOR function 235
 FLOORZ function 236
 formats 69
 categories of 73
 date, time, and datetime 54
 examples 71
 outputting formatted data 71
 specifying, changing, and deleting 70
 syntax 69
 validation of 71
 FRACTw. format 120
 FROM clause
 usage of CONNECTION TO
 component 59
 functions 179
 aggregate 179, 182
 ANSI Standard for SQL and 504
 calling Base SAS functions instead of
 FedSQL aggregate functions 183
 categories of 183
 date, time, and datetime 54
 DS2 package expressions as 181, 504
 FCMP procedure functions 504
 in expressions 341
 restrictions on arguments 181
 set functions 179
 syntax 180
 futures
 call prices for European options on,
 Black model 199
 put prices for European options on,
 Black model 201

G

GAMMA function 237
 GCD function 238
 GEOMEAN function 239
 GEOMEANZ function 241
 GP_DISTRIBUTED_BY= table option 459

H

HARMEAN function 242
 HARMEANZ function 243
 HASH= table option 460
 heterogeneous joins 40
 HEXw. format 121
 HHMMw.d format 122
 HOUR function 245
 HOURw.d format 124

I

identifiers 17
 delimited 17, 18
 overview 17
 regular 17
 support for non-Latin characters 18
 IDXNAME= table option 461
 IDXWHERE= table option 462
 IEEEw.d format 125
 IFNULL function 246
 IN predicate 329
 informats 345
 examples 347
 FedSQL support of 346
 specifying 347
 syntax 345
 validation of 347
 INSERT statement 382
 integrity constraints
 ANSI Standard for SQL and 504
 IOBLOCKSIZE= table option 463
 IQR function 247
 IS FALSE predicate 330
 IS MISSING predicate 331
 IS NULL predicate 333
 IS TRUE predicate 334
 IS UNKNOWN predicate 335

J

join operations 27
 JULDATE function 248
 JULDATE7 function 250
 JULIANw. format 126

K

KURTOSIS function 252

L

LABEL= table option 464
 LARGEST function 251
 LCM function 253
 LIKE predicate 336
 LOCKTABLE= table option 465
 LOG function 254
 LOG10 function 256
 LOG2 function 255
 LOGBETA function 257
 logical expressions
 type conversion for 24
 LOWCASE function 258

M

MAD function 259
 MAKEDATE function 260
 MAKETIME function 260
 MAKETIMESTAMP function 261
 Margrabe model
 call prices for European options on
 stocks 263
 put prices for European options on
 stocks 265
 MARGRCLPRC function 263
 MARGRPTPRC function 265
 MAX function 267
 MEAN function 269
 MEDIAN function 270
 MIN function 271
 MINUTE function 272
 missing values
 ANSI Standard for SQL and 504
 MMDDYYw. format 127
 MMDDYYxw. format 129
 MMSSw.d format 130
 MMYW. format 131
 MMYWxw. format 133
 MOD function 273
 MONNAMEw. format 134
 MONTH function 275
 MONTHw. format 135
 MONYYw. format 136

N

naming conventions
 SAS and SPD data sets 505
 NEGARENw.d format 137
 NENGOW. format 138
 NMISS function 276

non-Latin character support 18
 null values
 ANSI Standard for SQL and 504
 NULLIF expression 338
 numeric value expressions 50

O

OCTALw. format 140
 OCTET_LENGTH function 277
 operators 339
 order of evaluation 341
 options on futures
 call prices for European, based on Black
 model 199
 put prices for European, based on Black
 model 201
 options on stocks
 call prices for European, based on
 Black-Scholes model 203
 order of evaluation 341
 ORDER_BY= table option 466
 ORDINAL function 278
 ORHINTS= Table Option 467
 ORNUMERIC= table option 467
 output
 formatted data 71

P

PADCOMPRESS= table option 468
 PARTITION_KEY= table option 472
 PARTITION= table option 471
 PARTSIZE= table option 469
 PCTL function 279
 PERCENTNw.d format 142
 PERCENTw.d format 141
 PERM= table option 472
 PI function 281
 POINTOBS table option 473
 POWER function 281
 predicates 321
 order of evaluation 341
 probabilities
 binomial distributions 282
 chi-squared distributions 284
 probability 283
 PROBBNML function 282
 PROBBNRM function 283
 PROBCHI function 284
 PROBT function 285
 PROC FEDSQL
 data source connection 12
 PROC FEDSQL and FedSQL 10
 programs
 submitting 10

PUT function 286
 outputting formatted data 71
 put prices
 European options on futures, Black
 model 201
 for European options, based on
 Margrabe model 265
 PW= table option 474

Q

QTR function 288
 QTRRw. format 144
 QTRw. format 143
 queries 42
 correlated subqueries 45
 subqueries 44, 341
 query expressions 42
 QUOTE function 289

R

RADIANS function 290
 RANGE function 291
 READ= table option 474
 regular identifiers 17
 relational expressions
 type conversion for 25
 REPEAT function 292
 reserved words 61
 REUSE= table option 475
 REVERSE function 293
 RMS function 294
 ROLLBACK statement 385
 ROMANw. format 145
 row value constructors 51
 row value expressions 51

S

SAS data sets
 date, time, and datetime formats for 54
 dates and times in 52
 identifiers 505
 naming conventions 505
 SAS Federation Server
 data source connection 12
 SAS Federation Server and FedSQL 10
 SAS Federation Server LIBNAME engine
 and FedSQL 10
 scalar subqueries 44
 SECOND function 295
 SELECT statement 386
 set functions 179
 SIGN function 296
 SIN function 297

- SINH function 298
 - SIZEKMGw.d format 146
 - SIZEKw.d format 145
 - SKEWNESS function 299
 - SMALLEST function 300
 - SPD data sets
 - identifiers 505
 - naming conventions 505
 - SPD Engine data sets
 - date, time, and datetime formats for 54
 - dates and times in 52
 - SPD Server
 - data source connection 12
 - SPD Server and FedSQL 11
 - sql-expression 339
 - SQRT function 301
 - SQUEEZE= table option 476
 - STARTOBS= table option 476
 - statement table options
 - See [table options](#)
 - statements 349
 - STD function 302
 - STDDEV function 303
 - STDERR function 304
 - stocks
 - call prices for European options on, Black-Scholes model 203
 - call prices for European options, based on Margrabe model 263
 - put prices for European options, based on Margrabe model 265
 - STUDENTS_T function 305
 - subqueries 44
 - correlated 45
 - in expressions 341
 - scalar 44
 - SUBSTRING function 307
 - SUM function 308
 - syntax conventions 5
- T**
- table options 417
 - ANSI Standard for SQL and 504
 - interactions with other options 417
 - restrictions 417
 - syntax 418
 - TABLE_TYPE= table option 477
 - TAN function 309
 - TANH function 310
 - TD_BUFFER_MODE= table option 478
 - TD_CHECKPOINT= table option 478
 - TD_DATA_ENCRYPTION= table option 479
 - TD_DROP_ERROR_TABLE= table option 479
 - TD_DROP_LOG_TABLE= table option 480
 - TD_DROP_WORK_TABLE= table option 480
 - TD_ERROR_LIMIT= table option 481
 - TD_ERROR_TABLE_1= table option 481
 - TD_ERROR_TABLE_2= table option 482
 - TD_LOG_MECH_DATA= table option 484
 - TD_LOG_MECH_TYPE= table option 483
 - TD_LOG_TABLE= table option 485
 - TD_LOGDB= table option 485
 - TD_MAX_SESSIONS= table option 486
 - TD_MIN_SESSIONS= table option 487
 - TD_NOTIFY_LEVEL= table option 487
 - TD_NOTIFY_METHOD= table option 488
 - TD_NOTIFY_STRING= table option 489
 - TD_PACK_MAXIMUM= table option 490
 - TD_PACK= table option 489
 - TD_PAUSE_ACQ= table option 491
 - TD_SESSION_QUERY_BAND= table option 491
 - TD_TENACITY_HOURS= table option 492
 - TD_TENACITY_SLEEP= table option 492
 - TD_TPT_OPER= table option 493
 - TD_TRACE_LEVEL_INF= table option 494
 - TD_TRACE_LEVEL= table option 494
 - TD_TRACE_OUTPUT= table option 495
 - TD_WORK_TABLE= table option 496
 - TD_WORKING_DB= table option 496
 - THREADNUM= table option 497
 - TIMEAMPW.d format 149
 - TIMEPART function 311
 - times
 - See [dates and times](#)
 - TIMEw.d format 148
 - TODAY function 312
 - TODw.d format 151
 - transactions 61
 - TRIM function 313
 - type conversion 23
 - arithmetic expressions 25
 - concatenation expressions 26
 - definitions 23
 - logical expressions 24
 - overview 23

- relational expressions 25
- unary expressions 24
- TYPE= table option 498
- typographical conventions 5

U

- UCA= table option 498
- unary expressions
 - type conversion for 24
- UNIQUESAVE= table option 499
- UPCASE function 314
- UPDATE statement 412
- user privileges 505
- USS function 315

V

- validation of formats 71
- validation of informats 347
- value expressions 50
 - numeric 50
 - row value 51
- VARIANCE function 316
- VAXRBw.d format 152

W

- w.d format 153

- WEEKDATEw. format 154
- WEEKDATXw. format 155
- WEEKDAY function 317
- WEEKDAYw. format 157
- WHEREINDEX= table option 499
- WRITE= table option 500

Y

- YEAR function 318
- YEARw. format 158
- YENw.d format 159
- YYMMDDw. format 163
- YYMMDDxw. format 164
- YYMMw. format 160
- YYMMxw. format 161
- YYMONw. format 166
- YYQ function 319
- YYQRw. format 170
- YYQRxw. format 171
- YYQw. format 167
- YYQxw. format 168
- YYQZw. format 173

Z

- Zw.d format 174

