

# CCL Programmers Guide Sybase Event Stream Processor 5.0

#### DOCUMENT ID: DC01612-01-0500-04

LAST REVISED: April 2012

Copyright © 2012 by Sybase, Inc. All rights reserved.

This publication pertains to Sybase software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

Upgrades are provided only at regularly scheduled software release dates. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.

Sybase trademarks can be viewed at the Sybase trademarks page at *http://www.sybase.com/detail?id=1011207*. Sybase and the marks listed are trademarks of Sybase, Inc. <sup>®</sup> indicates registration in the United States of America.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world.

Java and all Java-based marks are trademarks or registered trademarks of Oracle and/or its affiliates in the U.S. and other countries.

Unicode and the Unicode Logo are registered trademarks of Unicode, Inc.

All other company and product names mentioned may be trademarks of the respective companies with which they are associated.

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

# Contents

CHAPTER 1: Introduction to Sybase Event Stream Processor	
Event Streams	
Event Stream Processor Compared to Databases	
Data-Flow Programming	
ESP Projects: Adapters, Streams, Windows, and	
Continuous Queries	
Streams Versus Windows	
Schemas	
Inserts, Updates, and Deletes	
Product Components	
Input and Output Adapters	
Custom Adapters	
Authoring Methods	
Continuous Computation Language	
SPLASH	10
CHAPTER 2: CCL Project Basics	11
Order of Elements	11
Comparing Streams, Windows, and Delta Streams	
Implicit Columns	
Input/Output/Local	
Windows	
Named Windows	
Unnamed Windows	
Retention	
Delta Streams	
Schemas	
Stores	

Working with Adapters	
Receiving Data with Input Adapters	
Publishing Data with Output Adapters	20
CHAPTER 3: Developing a Project in CCL	21
CHAPTER 4: Language Components	23
Datatypes	23
Intervals	
Operators	27
Expressions	31
CCL Comments	32
Case-Sensitivity	33
	05
CHAPTER 5: CCL Query Construction	
Filtering	
Unions	
Example: Merging Data from Streams or Windows	
Joins	
Key Field Rules	
Join Examples: ANSI Syntax	
Join Example: Comma-Separated Syntax	
Aggregation	
Ayyreyallon	43
CHAPTER 6: Advanced CCL Programming	
Techniques	45
•	
Declare Blocks	-
Typedefs Parameters	
Variables	48

Declaring Project Variables, Parameters, Datatypes,	
and Functions	50
Flex Operators	50
Modularity	51
Module Creation and Usage	52
Example: Creating and Using Modules	. 53
Example: Parameters in Modules	54
Data Recovery	55
Log Store Optimization Techniques	57
Error Streams	57
Monitoring Streams for Errors	59
CHAPTER 7: Statements	61
ADAPTER START Statement	
ATTACH ADAPTER Statement	
CREATE DELTA STREAM Statement	-
CREATE ERROR STREAM Statement	
CREATE FLEX Statement	
CREATE LIBRARY Statement	
CREATE LOG STORE Statement CREATE MEMORY STORE Statement	
CREATE MODULE Statement	
CREATE SCHEMA Statement	
DECLARE Statement	
CREATE STREAM Statement	
CREATE WINDOW Statement	
IMPORT Statement	
LOAD MODULE Statement	
	04
	~-
CHAPTER 8: Clauses	.87
AGING Clause	87
AS Clause	88
CASE Clause	89
IN Clause	90

KEEP Clause	91
OUT Clause	93
PARAMETERS Clause	94
PRIMARY KEY Clause	95
SCHEMA Clause	97
STORE Clause	98
STORES Clause	98
CHAPTER 9: Queries	101
FROM Clause	103
FROM Clause: Comma-Separated Syntax	103
FROM Clause: ANSI Syntax	104
GROUP BY Clause	106
GROUP FILTER Clause	107
GROUP ORDER BY Clause	108
HAVING Clause	109
MATCHING Clause	110
ON Clause: Join Syntax	112
SELECT Clause	113
UNION Operator	114
WHERE Clause	116
CHAPTER 10: Functions	119
User-Defined SPLASH Functions	119
User-Defined External Functions	120
External C/C++ Function Requirements	120
Example: Using External C/C++ Functions	122
Example: Using Java Functions	124
Aggregate Functions	126
any()	127
avg()	127
corr()	128
covar_pop()	129
covar_samp()	130

count()	130
count(distinct)	131
exp_weighted_avg()	131
first()	132
first_value()	133
last()	
last_value()	
lwm_avg()	133
max()	
meandeviation()	135
median()	
min()	
nth()	136
recent()	137
regr_avgx()	137
regr_avgy()	138
regr_count()	139
regr_intercept()	139
regr_r2()	140
regr_slope()	140
regr_sxx()	141
regr_sxy()	142
regr_syy()	142
stddev()	143
stddeviation()	143
stddev_pop()	
stddev_samp()	
sum()	144
valueinserted()	145
var_pop()	
var_samp()	146
vwap()	
weighted_avg()	147
xmlagg()	
Scalar Functions	148

Numeric Functions	149
acos()	149
asin()	149
atan()	150
atan2()	150
avgof()	151
bitand()	151
bitclear()	152
bitflag()	152
bitflaglong()	153
bitmask()	153
bitmasklong()	154
bitnot()	154
bitor()	154
bitset()	155
bitshiftleft()	155
bitshiftright()	156
bittest()	156
bittoggle()	157
bitxor()	157
cbrt()	158
ceil()	158
compare()	159
cos()	159
cosd()	160
cosh()	160
distance()	160
distancesquared()	161
floor()	162
isnull()	162
length()	163
ln()	163
log2()	164
log10()	164
logx()	165

maxof()16	65
minof()16	65
nextval()16	66
pi()16	66
power()16	67
random()16	67
round()	67
sign()	68
sin()	68
sinh()16	69
sqrt()16	69
tan()	69
tand()17	70
tanh()17	70
String Functions	
int32()17	
left()	
like()17	
lower()17	72
ltrim()	73
patindex()17	
real()17	
regexp_firstsearch()17	
regexp_replace()	
regexp_search()17	
replace()17	
right()	
rtrim()17	
string()17	
substr()17	
trim()	
trunc()17	
upper()18	
Conversion Functions	
ascii()	

base64_binary()	181
base64_string()	181
cast()	182
char()	183
concat()	183
extract()	184
fromnetbinary()	184
hex_binary()	185
hex_string()	185
msecToTime()	186
secToTime()	186
timeToMsec()	
timeToUsec()	187
timeToSec()	187
to_binary()	
to_bigdatetime()	188
to_boolean()	189
to_date()	190
to_float()	190
to_integer()	191
to_interval()	191
to_long()	192
to_money()	192
to_xml()	193
tonetbinary()	193
to_string()	194
to_timestamp()	196
usecToTime()	196
XML Functions	197
xmlconcat()	197
xmlelement()	197
xmlparse()	
xmlserialize()	
Date and Time Functions	199
business()	199

businessday()	199
date()	200
dateceiling()	200
datefloor()	202
datename()	203
datepart()	204
dateround()	204
dayofmonth()	206
dayofweek()	206
dayofyear()	207
hour()	207
intdate()	208
makebigdatetime()	208
microsecond()	210
minute()	210
month()	
second()	211
sysdate()	212
systimestamp()	212
unbigdatetime()	212
undate()	213
weekendday()	
year()	
Other Functions	214
cacheSize()	214
coalesce()	216
dateint()	216
deleteCache()	217
exp()	218
firstnonnull()	219
get*columnbyindex()	219
get*columnbyname()	220
getCache()	
getData()	222
getmoneycolumnbyindex()	223

Contents

getmoneycolumnbyname()	224
getrowid()	225
now()	
rank()	
recordDataToRecord	
recordDataToString	226
sind()	227
sysbigdatetime()	227
totimezone()	228
	220

# 

<b>APPENDIX B: Date and Time Programming.</b>	231
Time Zones	231
Changes to Time Zone Defaults	232
List of Time Zones	232
Date/Time Format Codes	240
Calendar Files	

 :47
2

# CHAPTER 1 Introduction to Sybase Event Stream Processor

Sybase<sup>®</sup> Event Stream Processor enables you to create and run your own complex event processing (CEP) applications to derive continuous intelligence from streaming event data in real time.

#### Event Stream Processing and CEP

Event stream processing is a form of CEP, a technique for analyzing information about events, in real time, for situational awareness. When vast numbers of event messages are flooding in, it is difficult to see the big picture. With event stream processing, you can analyze events as they stream in and identify emerging threats and opportunities as they happen. Event Stream Processor Server filters, aggregates, and summarizes data to enable better decision making based on more complete and timely information.

Event Stream Processor is not an end-user application, but an enabling technology that provides tools that make it easy to develop and deploy both simple and complex projects. It provides a highly scalable runtime environment in which to deploy those projects.

#### Event Stream Processor as a Development Platform

As a platform for developing CEP projects, Event Stream Processor provides high-level tools for defining how events are processed and analyzed. Developers can work in either a visual or text-oriented authoring environment. You can define logic that is applied to incoming events to:

- Combine data from multiple sources, producing derived event streams that include richer and more complete information.
- Compute value-added information to enable rapid decision making.
- Watch for specific conditions or patterns to enable instantaneous response.
- Produce high-level information, such as summary data, statistics, and trends to see the big picture, or the net effect, of many individual events.
- Continuously recompute key operating values based on complex analysis of incoming data.
- Collect raw and result data into a historical database for historical analysis and compliance.

#### Event Stream Processor Runtime Environment

As an engine for an event-driven architecture (EDA), Event Stream Processor can absorb, aggregate, correlate, and analyze events to produce new high-level events that can trigger responses, and high-level information that shows the current state of the business. Event Stream Processor:

- Processes data continuously as it arrives
- Processes data before it is stored on disk, thus achieving extremely high throughput and low latency, enabling better decision making based on more complete and timely information
- Separates business logic from data management, making it easier to maintain the business logic and reducing total cost of ownership
- Provides enterprise class scalability, reliability, and security

# **Event Streams**

An business event is a message that contains information about an actual business event that occurred. Many business systems produce events when things happen.

Examples of business events that are often transmitted as streams of event messages include:

- Financial market data feeds that transmit trade and quote events. Events may consist of ticket symbol, price, quantity, time, and so on
- Radio Frequency Identification System (RFID) sensors that transmit events indicating that an RFID tag was sensed nearby
- Click streams, which transmit a message (a click event) each time a user clicks a link, button, or control on a Web site
- Transaction events, which occur each time a record is added to a database or updated in a database

Many applications are already designed to produce events in real time, typically publishing them on a message bus. Applications that are not designed in this way can be "event enabled" using tools such as Sybase Replication Server<sup>®</sup>, which can monitor transaction logs to produce a real-time stream of events based on application database updates .

# **Event Stream Processor Compared to Databases**

Sybase Event Stream Processor complements traditional databases to help solve new classes of problems where continuous, event-driven data analysis is required.

Event Stream Processor is not a replacement for databases. Databases excel at storing and querying static data, and reliably processing transactions. However, databases are not effective at continuously analyzing fast moving streams of data.

- Traditional databases must store all data on disk before beginning to process it.
- Databases do not use preregistered continuous queries. Database queries are "one-timeonly" queries. To ask a question ten times a second, you must issue the query ten times a second. This model breaks down when one or more such queries need to be executed continuously as polling the database faster results in a performance impact to the source systems. Additionally, the polling approach has latency.

• Databases do not use incremental processing. Event Stream Processor can evaluate queries incrementally as data arrives.

Event Stream Processor is not an in-memory database. It shares some traits with in-memory databases in that it operates in and holds all data in memory, to achieve desired speed. However, unlike an in-memory database, that is designed to efficiently process on-demand queries, Event Stream Processor uses a data-flow architecture that is optimized for continuous event-driven queries.

# **Data-Flow Programming**

In data-flow programming, you define a set of event streams and the connections between them, and apply operations to the data as it flows from sources to outputs.

Data-flow programming breaks a potentially complex computation into a sequence of operations with data flowing from one operation to the next. This technique also provides scalability and potential parallelization, since each operation is event driven and independently applied. Each operation processes an event only when it is received from another operation. No other coordination is needed between operations.

# vwap.cd fxArb.cd rest vwap.cd fxArb.cd bbpc2.cd PortfolioValuation:Diagram X v vwap.cd fxArb.cd vwap.cd fxArb.cd vwap.cd fxArb.cd valueSystem valueByBook valueByBook valueByBook

#### Figure 1: Data-Flow Programming

# ESP Projects: Adapters, Streams, Windows, and Continuous Queries

An ESP project defines a set of event streams, any other required datasources, and the business logic applied to incoming event data to produce results.

At its most basic level, a project consists of:

- **Input streams and windows** where the input data flows into the project. An input stream can receive incoming event data on an event-driven basis, and can also receive static or semistatic sets of data that are loaded once or periodically refreshed.
- Adapters connect an input stream or window to a datasource. Sybase Event Stream Processor includes a large set of built-in adapters as well as an SDK that you can use to build custom adapters. Adapters can also connect an output stream or window to a destination.
- **Derived streams and windows** take data from one or more streams or windows and apply a continuous query to produce a new stream or window.

#### Getting Results from an ESP Project

Event Stream Processor has four ways to get output from a running project:

- Applications receive information automatically from internal output adapters attached to a stream when you build the project.
- Applications can subscribe to data streams by means of an external subscriber, which users can create using subscription APIs provided with the product.
- Users can start a new project that binds (connects) to a stream in a running project, without reconfiguring the project.
- Users can run on-demand queries using the esp\_query tool to query output windows in a running ESP project. For more information see the *Utilities Guide*.

#### See also

• Working with Adapters on page 19

#### **Streams Versus Windows**

Both streams and windows process events. The difference is that windows have state, meaning they can retain and store data, while streams are stateless and cannot.

Streams process incoming events and produce output events according to the continuous query that is attached to the stream, but no data is retained.

By contrast, a window consists of a table where incoming events can add rows, update existing rows, or delete rows. You can set the size of the window based on time, or on the number of

events recorded. For example, a window might retain all events over the past 20 minutes, or the most recent 1,000 events. A window can also retain all events. In this case, the incoming event stream must be self-managing in that it contains events that both insert rows into the window and delete rows from the window, so that the window does not grow infinitely large. Windows are needed for performing aggregate operations, as this cannot be done on streams.

#### Input, Output, and Local Streams and Windows

Streams and windows can be designated as input, output, or local. Input streams are the point at which data enters the project from external sources via adapters. A project may have any number of input streams. Input streams do not have continuous queries attached to them, although you can define filters for them.

Local and output streams and windows take their input from other streams or windows, rather than from adapters, and they apply a continuous query to produce their output. Local streams and windows are identical to output streams and windows, except that local streams and windows are hidden from outside subscribers. Thus, when a subscriber selects which stream or window to subscribe to, only output streams and windows are available.

**Note:** The visual authoring palette lists local and output streams as derived streams, and lists local and output windows as derived windows.

#### See also

• Comparing Streams, Windows, and Delta Streams on page 11

# Schemas

Each stream or window has a schema, which defines the columns in the events produced by the stream or window.

Each column has a name and datatype. All events that output from a single stream or window have an identical set of columns. For example:

- An input stream called RFIDRaw, coming out of an RFID reader, may have columns for a ReaderID and a TagID, both containing string data.
- An input stream called Trades, coming from a stock exchange, may have columns for the Symbol (string), Volume (integer), Price (float), and Time (datetime).

#### See also

- Schemas on page 17
- *Receiving Data with Input Adapters* on page 19
- Publishing Data with Output Adapters on page 20

#### Inserts, Updates, and Deletes

Operation Codes (opcodes) associate insert, update, and delete events with a window. They simplify development and improve performance by applying these events automatically.

In many Event Stream Processor use cases, events are independent of each other: each carries information about something that happened. In these cases, a stream of events is a series of independent events. If you define a window on this type of event stream, each incoming event is inserted into the window. If you think of a window as a table, the new event is added to the window as a new row.

In other use cases, events deliver new information about previous events. The ESP Server needs to maintain a current view of the set of information as the incoming events continuously update it. Two common examples are order books for securities in capital markets, or open orders in a fulfillment system. In both applications, incoming events may indicate the need to:

- Add an order to the set of open orders,
- Update the status of an existing open order, or,
- Remove a cancelled or filled order from the set of open orders.

To handle information sets that are updated by incoming events, Event Stream Processor recognizes insert, update, and delete operations associated with incoming events. You can tag events with an opcode, a special field that indicates whether the event is an insert event, an update event, or a delete event. There is also an upsert opcode, which either updates an existing record with a matching key, or inserts a new record.

Input windows apply insert, update, and delete events to the data in the window directly, as events arrive. Inserts, updates, and deletes are propagated through the query graph, that is, all downstream derived windows. Thus, when an event updates or deletes a record in an input window, it automatically applies to any downstream derived windows. This native handling of updates and deletes provides high performance and simplicity. Users do not need to manually define the logic to examine incoming events and determine how to apply them to a window.

# Product Components

Event Stream Processor includes a server component for processing and correlating streams of data, a Studio environment for developing, testing, and starting applications that run on the server, and administrative tools.

Components include:

• **ESP Server** – the software that processes and correlates data streams at runtime. Event Stream Processor can process and analyze hundreds of thousands of messages per second. Clustering provides scale-out support to ESP Server. A server cluster lets users run

multiple projects simultaneously; provides high availability and failover; and lets you apply centralized security and support for managing cluster connections.

- **ESP Studio** an integrated development environment for creating, modifying, and testing ESP projects.
- **CCL compiler** the compiler that translates and optimizes projects for processing by ESP Server. It is invoked by ESP Studio or from the command line.
- **Input and output adapters** the components that establish connections between Event Stream Processor and datasources, as well as the connections between the ESP Server and the consumers that will receive output from Event Stream Processor.
- **Integration SDK** a set of APIs for creating custom adapters in C/C++, Java, and .NET, for integrating custom function libraries, and for managing and monitoring live projects.
- Utilities a set of executables that offer command line access to many administrative, project development, publishing and subscription, and other features.

# **Input and Output Adapters**

Input and output adapters enable Event Stream Processor to send and receive messages from dynamic and static external sources and destinations.

External sources or destinations can include:

- Data feeds
- Sensor devices
- Messaging systems
- Radio frequency identification (RFID) readers
- E-mail servers
- Relational databases

Input adapters connect to an external datasource and translate incoming messages from the external sources into a format that is accepted by the ESP server. Output adapters translate rows processed by Event Stream Processor into message formats that are compatible with external destinations and send those messages downstream.

The following illustration shows a series of input adapters that translate messages from a temperature sensor, bar code scanner, and a Java Message Service (JMS) cloud into formats compatible with Event Stream Processor. After the data is processed using various queries within Event Stream Processor, output adapters convert the result rows into updates that are sent to an external database server, e-mail server, and Web services dashboard.



Figure 2: Adapters in Event Stream Processor

For a complete list of adapters supplied by Event Stream Processor, see the *Adapters Guide*.

#### **Custom Adapters**

In addition to the adapters provided by Event Stream Processor, you can write your own adapters to integrate into the server.

Event Stream Processor provides a variety of SDKs that allow you to write adapters in a number of programming languages, including:

- C
- C++
- Java
- .NET (C#, Visual Basic, and so on)

For detailed information about how to create custom adapters, see the *Adapters Guide*. For versions supported by these SDKs, see the *Installation Guide*.

# **Authoring Methods**

Event Stream Processor Studio provides visual and text authoring environments for developing projects.

In the visual authoring environment, you can develop projects using graphical tools to define streams and windows, connect them, integrate with input and output adapters, and create a project consisting of queries.

In the text authoring environment, you can develop projects in the Continuous Computation Language (CCL), as you would in any text editor. Create data streams and windows, develop queries, and organize them in hierarchical modules and projects.

You can easily switch between the Visual editor and the CCL editor at any time. Changes made in one editor are reflected in the other. You can also compile projects within Studio.

In addition to its visual and text authoring components, Studio includes environments for working with sample projects, and for running and testing applications with a variety of debugging tools. Studio also lets you record and playback project activity, upload data from files, manually create input records, and run ad hoc queries against the server.

If you prefer to work from the command line, you can develop and run projects using the **esp\_server**, **esp\_client**, and **esp\_compiler** commands. For a full list of Event Stream Processor utilities, see the *Utilities Guide*.

#### **Continuous Computation Language**

CCL is the primary event processing language of the Event Stream Processor. ESP projects are defined in CCL.

CCL is based on Structured Query Language (SQL), adapted for event stream processing.

CCL supports sophisticated data selection and calculation capabilities, including features such as: data grouping, aggregations, and joins. However, CCL also includes features that are required to manipulate data during real-time continuous processing, such as windows on data streams, and pattern and event matching.

The key distinguishing feature of CCL is its ability to continuously process dynamic data. A SQL query typically executes only once each time it is submitted to a database server and must be resubmitted every time a user or an application needs to reexecute the query. By contrast, a CCL query is continuous. Once it is defined in the project, it is registered for continuous execution and stays active indefinitely. When the project is running on the ESP Server, a registered query executes each time an event arrives from one of its datasources.

Although CCL borrows SQL syntax to define continuous queries, the ESP server does not use an SQL query engine. Instead, it compiles CCL into a highly efficient byte code that is used by the ESP server to construct the continuous queries within the data-flow architecture.

CCL queries are converted to an executable form by the CCL compiler. ESP servers are optimized for incremental processing, hence the query optimization is different than for databases. Compilation is typically performed within Event Stream Processor Studio, but it can also be performed by invoking the CCL compiler from the command line.

# SPLASH

Stream Processing LAnguage SHell (SPLASH) is a scripting language that brings extensibility to CCL, allowing you to create custom operators and functions that go beyond standard SQL.

The ability to embed SPLASH scripts in CCL provides tremendous flexibility, and the ability to do it within the CCL editor maximizes user productivity. SPLASH also allows you to define any complex computations that are easier to define using procedural logic rather than a relational paradigm.

SPLASH is a simple scripting language comprised of expressions used to compute values from other values, as well as variables, and looping constructs, with the ability to organize instructions in functions. SPLASH syntax is similar to C and Java, though it also has similarities to languages that solve relatively small programming problems, such as AWK or Perl.

#### See also

- Declare Blocks on page 45
- User-Defined SPLASH Functions on page 119
- CREATE FLEX Statement on page 67

# CHAPTER 2 CCL Project Basics

Before you begin a project, be familiar with the order in which statements and clauses must appear, project development, and the basics of streams and windows.

#### **Order of Elements**

Determine the order of CCL project elements based on clause and statement syntax definitions and limitations.

Define CCL elements that are referenced by other statements or clauses before using those statements and clauses. Failure to do so causes compilation errors.

For example, define a schema using a **CREATE SCHEMA** statement before a CCL **CREATE STREAM** statement references that schema by name. Similarly, declare parameters and variables in a declare block before any CCL statements or clauses reference those parameters or variables.

You cannot reorder subclause elements within CCL statements or clauses.

#### **Comparing Streams, Windows, and Delta Streams**

Streams, windows, and delta streams offer different characteristics and features, but also share common designation, visibility, and column parameters.

The terms "stateless" and "stateful" commonly describe the most significant difference between windows and streams. A stateful element has the capacity to store information, while a stateless element does not.

Feature Capabili- ty	Streams	Windows	Delta Streams
Type of element	Stateless	Stateful, due to reten- tion and store capabili- ties	Stateless
Data retention	None	Yes, rows (based on re- tention policy)	None
Available store types	Not applicable	Memory store or log store	Not applicable

Feature Capabili- ty	Streams	Windows	Delta Streams
Element types that can be derived from this el- ement	Stream or a Window with an aggregation clause (GROUP BY)	Stream, Window, Delta Stream	Stream, Window, Delta Stream
Primary key Required	No	Yes, explicit or de- duced	Yes, explicit or de- duced
Support for aggrega- tion operations	No	Yes	No
Behavior on receiving update	Receives and produces insert	Receives and produces update	Receives and produces update
Behavior on receiving insert	Receives and produces insert	Receives and produces insert	Receives and produces insert
Behavior on receiving <b>delete</b>	Receives but ignores	Receives and produces delete	Receives and produces delete

Streams, windows, and delta streams share several important characteristics, including implicit columns and visibility rules.

#### See also

- Windows on page 14
- Streams Versus Windows on page 4

#### **Implicit Columns**

All streams, windows, and delta streams use three implicit columns called ROWID, ROWTIME, and BIGROWTIME.

- ROWID provides a unique row identification number for each row of incoming data.
- ROWTIME provides the last modification time as a date with second precision.
- BIGROWTIME provides the last modification time of the row with microsecond precision. You can perform filters and selections based on these columns, like filtering out all of those data rows that occur outside of business hours.

#### Input/Output/Local

You can designate streams, windows, and delta streams as input, output, or local.

#### Input/Output Streams and Windows

Input streams and windows can accept data from a source external to the project using an input adapter or by connecting to an external publisher. You can attach an output adapter or connect external subscribers directly to an input window or input stream. You can also use the SQL

interface to SELECT rows from an input window, INSERT rows in an input stream or INSERT/ UPDATE/DELETE rows in an input window.

Output windows, streams and delta streams can publish data to an output adapter or an external subscriber. You can use the SQL interface to query (that is **SELECT**) rows from an output window.

Local streams, windows, and delta streams are invisible outside the project and cannot have input or output adapters attached to them. You cannot subscribe to or use the SQL interface to query the contents of local streams, windows, or delta streams.

#### Examples

This is an input stream with a filter:

CREATE SCHEMA mySchema (Coll INTEGER, Col2 STRING); CREATE INPUT STREAM IStr2 SCHEMA mySchema WHERE IStr2.Col2='abcd';

This is an output stream:

CREATE OUTPUT STREAM OStr1 AS SELECT A.Col1 col1, A.Col2 col2 FROM IStr1 A;

This is an input window:

CREATE SCHEMA mySchema (Coll INTEGER, Col2 STRING); CREATE MEMORY STORE myStore; CREATE INPUT WINDOW IWin1 SCHEMA mySchema PRIMARY KEY(Coll) STORE myStore;

This is an output window:

```
CREATE SCHEMA mySchema (Coll INTEGER, Col2 STRING);
CREATE MEMORY STORE myStore;
CREATE OUTPUT WINDOW OWin1
PRIMARY KEY (Coll)
STORE myStore
AS SELECT A.Coll coll, A.Col2 col2
FROM IWin1 A;
```

#### Local Streams and Windows

Use a local stream, window, or delta stream when the stream does not need an adapter, or to allow outside connections. Local streams, windows, and delta streams are visible only inside the containing CCL project, which allows for more optimizations by the CCL compiler. Streams and windows that do not have a qualifier are local.

**Note:** A local window cannot be debugged because it is not visible to the ESP Studio run/test tools such as viewer or debugger.

#### Examples

This is a local stream:

```
CREATE SCHEMA mySchema (Coll INTEGER, Col2 STRING);
CREATE LOCAL STREAM LStr1
AS SELECT i.Col1 col1, i.Col2 col2
FROM IStr1 i;
```

This is a local window:

```
CREATE SCHEMA mySchema (Coll INTEGER, Col2 STRING);
CREATE MEMORY STORE myStore;
CREATE LOCAL WINDOW LWin1
PRIMARY KEY (Col1)
STORE myStore
AS SELECT i.Col1 col1, i.Col2 col2
FROM IStr1 i;
```

#### Windows

A window is a stateful element that can be named or unnamed, and retains rows based on a defined retention policy.

#### See also

• Comparing Streams, Windows, and Delta Streams on page 11

#### **Named Windows**

A named window is explicitly created using a **CREATE WINDOW** statement, and can be referenced in other queries.

Named windows can be classed as input, output, or local. An input window can send and receive data through adapters. An output window can send data to an adapter. Both input and output windows are visible externally and can be subscribed to or queried. A local window is private and invisible externally. When a qualifier for the window is missing, it is presumed to be of type local.

Named Window Type	Receives Data From	Sends Data To	Visible Externally
input	Adapter or connector	Adapter or connector and other streams	Yes
output	Other streams	Adapter or connector and other streams	Yes
local	Other streams	Other streams	No

**Table 1. Named Window Capabilities** 

#### See also

• CREATE WINDOW Statement on page 80

#### **Unnamed Windows**

An unnamed window is an implicitly created stateful element that cannot be referenced or used elsewhere in a project.

An unnamed window is implicitly created when the **KEEP** clause is used with a source name in the **FROM** clause of a statement. Refer to the *KEEP Clause* on page 91 topic for details on the syntax.

**Note:** On a Delta Stream, only unnamed windows can be created by specifying the **KEEP** clause from the **FROM** clause.

#### Examples

This example creates an unnamed window on the input Trades for the MaxTradePrice window to keep track of a maximum trade price for all symbols seen within the last 10000 trades:

```
CREATE WINDOW MaxTradePrice
PRIMARY KEY DEDUCED
STORE S1
AS SELECT trd.Symbol, max(trd.Price) MaxPrice
FROM Trades trd KEEP 10000 ROWS
GROUP BY trd.Symbol;
```

This example creates an unnamed window on Trades, and MaxTradePrice keeps track of the maximum trade price for all the symbols during the last 10 minutes of trades:

In both examples, Trades can be a delta stream, or a window.

#### See also

- CREATE WINDOW Statement on page 80
- KEEP Clause on page 91

#### Retention

A retention policy specifies the maximum number of rows or the maximum period of time that data are retained in a window.

In CCL, you can specify a retention policy when defining a Window. You can also create an Unnamed Window by specifying a retention policy on a Window or Delta Stream when it is used as source to another element.

Retention is specified through the **KEEP** clause. You can limit the number of records in a window based on either the number, or age, of records in the window. These methods are referred to as count-based retention and time-based retention, respectively. If you do not specify a retention policy, the window retains all records. Or, you can use the **ALL** modifier to explicitly specify that the window should retain all records.

Specifying the **KEEP** clause with no modifier produces a Sliding Window, which deletes individual rows once a maximum age is reached or the maximum number of rows are retained.

**Note:** You can only specify retention on windows with memory-based stores. Retention on a log file-based store is only supported for input windows or when data is exactly copied from its source.

#### Count-based Retention

In a count-based policy, a constant integer specifies the maximum number of rows retained in the window. You can use parameters in the count expression.

A count-based policy also defines an optional SLACK value, which can enhance performance by requiring less frequent cleaning of memory stores. A SLACK value accomplishes this by ensuring that there are no more than N + S rows in the window, where N is the retention size and S is the SLACK value. When the window reaches N + S rows, the system purges S rows. The larger the SLACK value, the better the performance, since there is less cleaning required.

The default value for SLACK is 1, which means that after the window reaches the maximum number of records, every new record inserted deletes the oldest record. This causes a significant impact on performance. Larger slack values improve performance by reducing the need to constantly delete rows.

The following example creates a Sliding Window that retains the most recent 100 records that match the filter condition. Once there are 100 records in the window, the arrival of a new record causes the deletion of the oldest record in the window.

```
CREATE WINDOW Last100Trades PRIMARY KEY DEDUCED
KEEP 100 ROWS
AS SELECT * FROM Trades
WHERE Trades.Volume > 1000;
```

Adding the SLACK value of 10 means the window may contain as many as 110 records before any records are deleted.

```
CREATE WINDOW Last100Trades PRIMARY KEY DEDUCED
KEEP 100 ROWS SLACK 10
AS SELECT * FROM Trades
WHERE Trades.Volume > 1000;
```

#### Time-based Retention

In a time-based policy, a constant interval expression specifies the maximum age of the rows retained in the window.

The following example creates a Sliding Window that retains each record received for ten minutes. As each individual row exceeds the ten minute retention time limit, it is deleted.

```
CREATE WINDOW RecentPositions PRIMARY KEY DEDUCED
KEEP 10 MINS
AS SELECT * FROM Positions;
```

#### See also

- Data Recovery on page 55
- KEEP Clause on page 91

#### **Delta Streams**

Delta streams are stateless elements that can understand all opcodes.

You can use a delta stream anywhere you use a computation, filter, or union, but do not need to maintain a state. A delta stream performs these operations more efficiently than a window because it keeps no state, thereby reducing memory use and increasing speed.

#### Example

This example creates a delta stream named DeltaTrades that incorporates the **getrowid** and **now** functions.

```
CREATE LOCAL DELTA STREAM DeltaTrades
    SCHEMA (
        RowId long,
        Symbol STRING,
        Ts bigdatetime,
        Price MONEY(2),
        Volume INTEGER,
        ProcessDate bigdatetime )
    PRIMARY KEY (Ts)
AS SELECT getrowid ( TradesWindow) RowId,
        TradesWindow.Symbol,
         TradesWindow.Ts Ts,
         TradesWindow.Price,
         TradesWindow.Volume,
         now() ProcessDate
    FROM TradesWindow
CREATE OUTPUT WINDOW TradesOut
    PRIMARY KEY DEDUCED
AS SELECT * FROM DeltaTrades ;
```

#### Schemas

A schema defines the structure of data rows in a stream or window.

Every row in a stream or window must have the same structure, or schema, which includes the column names, the column datatypes, and the order in which the columns appear. Multiple streams or windows may use the same schema, but a stream or window can only have one schema.

Create a schema using the **CREATE SCHEMA** statement and associate it with a particular stream or window. You can create inline schemas within the syntax you use to create streams and windows, or create a named schema separately to reference later.

#### Simple Schema CCL Example

This is an example of a **CREATE SCHEMA** statement used to create a named schema. TradeSchema represents the name of the schema.

```
CREATE SCHEMA TradeSchema (
Ts BIGDATETIME,
Symbol STRING,
Price MONEY(4),
Volume INTEGER
);
```

This example uses a CREATE SCHEMA statement to make an inline schema:

```
CREATE STREAM trades SCHEMA (

Ts bigdatetime,

Symbol STRING,

Price MONEY(4),

Volume INTEGER
);
```

#### See also

• CREATE SCHEMA Statement on page 76

#### Stores

Set store defaults, or choose a log store or memory store to specify how data from a window is saved.

If you do not set a default store using the **CREATE DEFAULT STORE** statement, each window is assigned to a default memory store. You can use default store settings for store types and locations if you do not assign new windows to specific store types.

#### Memory Stores

A memory store does not use persistence and stores all data in memory. Memory stores retain the state of queries for a project from the most recent server start-up for as long as the project is running. Because query state is retained in memory rather than on disk, access to a memory store is faster than to a log store.

Use the **CREATE MEMORY STORE** statement to create memory stores. If no default store is defined, new windows are automatically assigned to a memory store.

#### Log Stores

Use a log store to implement persistence, which logs all data to the disk, meaning it guarantees data state recovery in the event of a failure.

Use the **CREATE LOG STORE** statement to create a log store. You can also set a log store as a default store using the **CREATE DEFAULT STORE** statement, which overrides the default memory store.

Log store dependency loops are a concern when using log stores, as they cause compilation errors. Log store loops can be created when you use multiple log stores in a project, and assign windows to these stores. The recommended way to use a log store is to either assign log stores to source windows only or to assign all windows in a stream path to the same store. If you use logstorel for n of those windows, then use logstore2 for a different window, you should never use logstore1 again further down the chain. Put differently, if Window Y assigned to Logstore B gets its data from Window X assigned to Logstore A, no window that (directly or indirectly) gets its data from Window Y should be assigned to Logstore A.

#### See also

- Data Recovery on page 55
- CREATE MEMORY STORE Statement on page 73
- CREATE LOG STORE Statement on page 71

# Working with Adapters

Determine the prerequisites for using Event Stream Processor-supplied adapters and custom adapters.

See the *Adapters Guide* for detailed information about configuring individual adapters, datatype mapping, and schema discovery.

#### See also

- ATTACH ADAPTER Statement on page 62
- CREATE SCHEMA Statement on page 76
- Parameters on page 47

#### **Receiving Data with Input Adapters**

Use an input adapter to pull information into the system from external data sources.

This is an overview of typical tasks to perform before attaching an input adapter to the ESP Server, as well as an introduction to the **ATTACH ADAPTER** statement.

- 1. Assess the input data. Determine which sets or subsets of data you want to pull into the ESP Server.
- 2. Choose an input adapter.

If the data source uses datatypes that are not supported by ESP Server, the Server maps the data to valid datatypes. Review the associated mapping description for your adapter in the *Adapters Guide*.

**3.** Configure your adapter.

- **4.** Create an input stream or window, and define the structure for the incoming data in its schema clause; alternatively use the schematic discovery feature in the visual editor to import the schematic and create an attached stream or window.
- **5.** Use the **ATTACH ADAPTER** statement to attach your adapter to the Server stream or window, and set values for the adapter properties.

The **ATTACH ADAPTER** statement includes parameters that allow adapter properties to be modified at server runtime.

**Note:** You can bind only declared parameters to a new value when a module or project is loaded.

Continue to build the model, assuming the input from the source is available through the stream or window you have defined.

#### Publishing Data with Output Adapters

Perform these tasks before you attach an output adapter to an external data destination.

- **1.** Assess the output data. Determine which sets or subsets of data to send to an external data destination.
- 2. Choose an output adapter.

If the output destination uses datatypes that are not supported by ESP Server, the Server maps the data to valid datatypes. Review the associated mapping description for your adapter in the *Adapters Guide*.

- 3. Configure your adapter.
- 4. Create an output stream or window and define the structure for outgoing data.
- **5.** Use the **ATTACH ADAPTER** statement to attach your adapter to the output stream or window, and set values for the adapter properties.

The **ATTACH ADAPTER** statement includes parameters that allow adapter properties to be modified at server runtime.

**Note:** You can bind only declared parameters to a new value when a module or project is loaded.

Data flows through the output stream or window and is sent by the adapter to the external data destination.

# CHAPTER 3 Developing a Project in CCL

Use the CCL Editor in ESP Studio, or another supported editor, to create and modify your CCL code. Start by developing a simple project, and test it iteratively as you gradually add greater complexity.

For details of these high-level steps, see the rest of this *CCL Programmers Guide*, as well as the *Studio Users Guide*, the *Adapters Guide*, and the *SPLASH Programmers Guide*.

1. Create a .ccl file.

Creating a project in ESP Studio creates the .ccl file automatically.

- 2. Add input streams and windows.
- 3. Add output streams and windows with simple continuous queries.
- 4. Attach adapters to streams and windows to subscribe to external sources or publish output.
- **5.** Compile the CCL code.
- **6.** Run the compiled project against test data, using the debugging tools in ESP Studio and command line utilities.

Repeat this step as often as needed.

- 7. Add queries to the project. Start with simple queries and gradually add complexity.
- **8.** (Optional) Use functions in your continuous queries to perform mathematical operations, aggregations, datatype conversions, and other common tasks:
  - Built-in functions for many common operations
  - User-defined functions written in the SPLASH programming language
  - User-defined external functions written in C/C++ or Java
- **9.** (Optional) Create named schemas to define a reusable data structure for streams or windows.
- **10.** (Optional) Create memory stores or log stores to retain the state of data windows in memory or on disk.
- **11.** (Optional) Create modules to contain reusable CCL that can be loaded multiple times in a project.

CHAPTER 3: Developing a Project in CCL

# CHAPTER 4 Language Components

To ensure proper language use in your CCL projects, familiarize yourself with rules on casesensitivity, supported datatypes, operators, and expressions used in CCL.

# **Datatypes**

Sybase Event Stream Processor supports integer, float, string, money, long, and timestamp datatypes for all of its components.

Datatype	Description
integer	A signed 32-bit integer. The range of allowed values is -2147483648 to +2147483647 ( $-2^{31}$ to $2^{31-1}$ ). Constant values that fall outside of this range are automatically processed as long datatypes.
	To initialize a variable, parameter, or column with a value of -2147483648, specify (-2147483647) -1 to avoid CCL compiler errors.
long	A signed 64-bit integer. The range of allowed values is -9223372036854775808 to +9223372036854775807 (-2 <sup>63</sup> to 2 <sup>63-1</sup> ).
	To initialize a variable, parameter, or column with a value of -9223372036854775808, specify (-9223372036854775807) -1 to avoid CCL compiler errors.
float	A 64-bit numeric floating point with double precision. The range of allowed values is approximately $-10^{308}$ through $+10^{308}$ .
string	Variable-length character string, with byte values encoded in UTF-8. Maximum string length is platform-dependent, but can be no more than 65535 bytes.
money	A legacy datatype maintained for backward compatibility. It is a signed 64-bit integer that supports 4 digits after the decimal point. Currency symbols and commas are not supported in the input data stream.

Datatype	Description	
money(n)	A signed 64-bit numerical value that supports varying scale, from 1 to 15 digits after the decimal point. Currency symbols and commas are not supported in the input data stream, however, decimal points are.	
	The supported range of values change, depending on the specified scale.	
	money(1):-922337203685477580.8 to 922337203685477580.7	
	money(2):-92233720368547758.08 to 92233720368547758.07	
	money(3):-9223372036854775.808 to 9223372036854775.807	
	money(4):-922337203685477.5808 to 922337203685477.5807	
	money(5):-92233720368547.75808 to 92233720368547.75807	
	money(6):-92233720368547.75808 to 92233720368547.75807	
	money(7):-922337203685.4775808 to 922337203685.4775807	
	money(8):-92233720368.54775808 to 92233720368.54775807	
	money(9):-9223372036.854775808 to 9223372036.854775807	
	money(10):-922337203.6854775808 to 922337203.6854775807	
	money(11):-92233720.36854775808 to 92233720.36854775807	
	money(12):-9223372.036854775808 to 9223,372.036854775807	
	money(13):-922337.2036854775808 to 922337.2036854775807	
	money(14):-92233.72036854775808 to 92233.72036854775807	
	money(15):-9223.372036854775808 to 9223.372036854775807	
	To initialize a variable, parameter, or column with a value of -92,233.72036854775807, specify (-97) -1 to avoid CCL compiler errors.	
	Specify explicit scale for money constants with <b>Dn</b> syntax, where <b>n</b> represents the scale. For example, 100.1234567D7, 100.12345D5.	
	Implicit conversion between money (n) types is not supported be- cause there is a risk of losing range or scale. Perform the <b>cast</b> function to work with money types that have different scale.	
Datatype	Description	
-------------	--	--
bigdatetime	Timestamp with microsecond precision. The default format is YYYY MM-DDTHH:MM:SS:SSSSSS.	
	All numeric datatypes are implicitly cast to bigdatetime.	
	The rules for conversion vary for some datatypes:	
	<ul> <li>All boolean, integer, and long values are converted in their original format to bigdatetime</li> <li>Only the whole-number portions of money(n) and float values are converted to bigdatetime. Use the cast function to convert money(n) and float values to bigdatetime with precision.</li> <li>All date values are multiplied by 1000000 and converted to microseconds to satisfy bigdatetime format.</li> <li>All timestamp values are multiplied by 1000 and converted to microseconds to satisfy bigdatetime format.</li> </ul>	
timestamp	Timestamp with millisecond precision. The default format is YYYY-	
	MM-DDTHH:MM:SS:SSS.	
date	Date with second precision. The default format is YYYY-MM-DDTHH:MM:SS.	

Datatype	Description	
interval	A signed 64-bit integer that represents the number of microseconds between two timestamps. Specify an interval using multiple units in space-separated format, for example, "5 Days 3 hours 15 Minutes". External data that is sent to an interval column is assumed to be in microseconds. Unit specification is not supported for interval values converted to or from string data.	
	When an interval is specified, the given interval must fit in a 64-bit integer (long) when it is converted to the appropriate number of microseconds. For each interval unit, the maximum allowed val- ues that fit in a long when converted to microseconds are:	
	<ul> <li>MICROSECONDS (MICROSECOND, MICROS): +/- 9223372036854775807</li> <li>MILLISECONDS (MILLISECOND, MILLIS): +/- 9223372036854775</li> <li>SECONDS(SECOND, SEC): +/- 9223372036854</li> <li>MINUTES(MINUTE, MIN): +/- 153722867280</li> <li>HOURS(HOUR,HR): +/- 2562047788</li> <li>DAYS(DAY): +/- 106751991</li> <li>The values in parentheses are alternate names for an interval unit. When the maximum value for a unit is specified, no other unit can be specified or it causes an overflow. Each unit can be specified only once.</li> </ul>	
binary	Represents a raw binary buffer. Maximum length of value is platform- dependent, but can be no more than 65535 bytes. NULL characters are permitted.	
boolean	Value is true or false. The format for values outside of the allowed range for boolean is 0/1/false/true/y/n/on/off/yes/no, which is case-insensitive.	

#### See also

• cast() on page 182

# Intervals

Interval syntax supports day, hour, minute, second, millisecond, and microsecond values.

Intervals measure the elapsed time between two timestamps, using 64 bits of precision. All occurrences of intervals refer to this definition:

value | {value [ {DAY[S] | {HOUR[S] | HR} | MIN[UTE[S]] | SEC[OND[S]] | {MILLISECOND[S] | MILLIS} | {MICROSECOND[S] | MICROS} ] [...]} If only value is specified, the timestamp default is MICROSECOND[S]. You can specify multiple time units by separating each unit with a space, however, you can specify each unit only once. For example, if you specify HOUR[S], MIN[UTE[S]], and SEC[OND[S]] values, you cannot specify these values again in the interval syntax.

Time Unit	Maximum Value Allowed
MICROSECOND[S]   MICROS	9,223,372,036,854,775,807
MILLISECOND[S]   MILLIS	9,233,372,036,854,775
SEC[OND[S]]	9,223,372,036,854,775
MIN[UTE[S]]	153,722,867,280,912
HOUR[S]   HR	2,562,047,788,015
DAY[S]	106,751,991,167

Each unit has a maximum value when not combined with another unit:

These maximum values decrease when you combine units.

Specifying value with a time unit means it must be a positive value. If value is negative, it is treated as an expression. That is, -10 MINUTES in the interval syntax is treated as - (10 MINUTES). Similarly, 10 MINUTES-10 SECONDS is treated as (10 MINUTES)-(10 SECONDS).

The time units can be specified only in CCL. When specifying values for the interval column using the API or adapter, only the numeric value can be specified and is always sent in microseconds.

#### Examples

```
3 DAYS, 1 HOUR, 54 MINUTES2 SECONDS, 12 MILLISECONDS, 1 MICROSECOND
```

# Operators

CCL supports a variety of numeric, nonnumeric, and logical operator types.

### Arithmetic Operators

Arithmetic operators are used to negate, add, subtract, multiply, or divide numeric values. They can be applied to numeric types, but they also support mixed numeric types. Arithmetic operators can have one or two arguments. A unary arithmetic operator returns the same datatype as its argument. A binary arithmetic operator chooses the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data-type, and returns that type.

#### CHAPTER 4: Language Components

Operator	Meaning	Example Usage
+	Addition	3+4
-	Subtraction	7-3
*	Multiplication	3*4
/	Division	8/2
%	Modulus (Remainder)	8%3
^	Exponent	4^3
-	Change signs	-3
++	Increment	++a (preincrement)
	Preincrement (++ <i>argument</i> ) value is incremented before it is passed as an argument Postincrement ( <i>argument</i> ++) value is passed and then in- cremented	a++ (postincre- ment)
	Decrement Predecrement ( <i>argument</i> ) value is decremented before it is passed as an argument Postdecrement ( <i>argument</i> ) value is passed and then de- cremented	a (predecrement) a (postdecrement)

#### **Comparison Operators**

Comparison operators compare one expression to another. The result of such a comparison can be TRUE, FALSE, or NULL.

Comparison operators use this syntax:

expression1 comparison\_operator expression2

Operator	Meaning	Example Us- age
=	Equality	a0=a1
!=	Inequality	a0!=a1
$\diamond$	Inequality	a0<>a1
>	Greater than	a0!>a1
>=	Greater than or equal to	a0!>=a1

Operator	Meaning	Example Us- age
<	Less than	a0! <a1< td=""></a1<>
<=	Less than or equal to	a0!<=a1
IN	Member of a list of values. If the value is in the expression list's values, then the result is TRUE.	a0 IN (a1, a2, a3)

### Logical Operators

Operator	Meaning	Example Usage
AND	Returns TRUE if all expressions are TRUE, and FALSE otherwise.	(a < 10) AND (b > 12)
NOT	Returns TRUE if all expressions are FALSE, and TRUE otherwise.	NOT (a = 5)
OR	Returns TRUE if any of the expressions are TRUE, and FALSE otherwise.	(b = 8) OR (b = 6)
XOR	Returns TRUE if one expression is TRUE and the other is FALSE. Returns FALSE if both expressions are TRUE or both are FALSE.	(b = 8) XOR (a > 14)

### String Operators

Operator	Meaning	Example Usage
+	Concatenates strings and returns another string.	'go' + 'cart'
	<b>Note:</b> The + operator does not support mixed datatypes (such as an integer and a string).	

### LIKE Operators

May be used in column expressions and **WHERE** clause expressions. LIKE supports the use of the LIKE and REGEXP\_LIKE operators to match string expressions to strings that closely resemble each other but do not exactly match.

Operator	Syntax and Meaning	Example Usage
LIKE	Matches <b>WHERE</b> clause string expressions to strings that closely resemble each other but do not exactly match.	Trades.StockName LIKE "%Corp%"
	compare_expression LIKE pat- tern_match_expression	
	The LIKE operator returns a value of TRUE if <b>compare_ex-pression</b> matches <b>pattern_match_expression</b> , or FALSE if it does not. The expressions can contain wildcards, where the percent sign (%) matches any length string, and the under-score (_) matches any single character.	

### [] Operator

The [] operator is only supported in the context of dictionaries and vectors.

Operator	Syntax and Meaning	Example Usage
	Allows you to perform functions on rows other than the cur- rent row in a stream or window. <pre>stream-or-window-name[index].column</pre> stream-or-window-name is the name of a stream or window and column indicates a column in the stream or window. index is an expression that can include literals, parameters, or operators, and evaluates to an integer. This integer indicates the stream or window row, in relation to the current row or to the window's sort order.	MyNamedWind- ow[1].MyColumn

### Order of Evaluation for Operators

When evaluating an expression with multiple operators, the engine evaluates operators with higher precedence before those with lower precedence. Those with equal precedence are evaluated from left to right within an expression. You can use parentheses to override operator precedence, since the engine evaluates expressions inside parentheses before evaluating those outside.

Note: The  $^{\circ}$  operator is right-associative. Thus, a  $^{\circ}$  b  $^{\circ}$  c = a  $^{\circ}$  (b  $^{\circ}$  c), not (a  $^{\circ}$  b)  $^{\circ}$  c.

The operators in order of preference are as follows. Operators on the same line have the same precedence:

- +.- (as unary operators)
- ^
- \*,/,%
- +, (as binary operators and for concatenation)
- =, !=, <>, <, >, <=, >= (comparison operators)

- LIKE, IN, IS NULL, IS NOT NULL
- NOT
- AND
- OR, XOR

# **Expressions**

An expression is a combination of one or more values, operators, and built in functions that evaluate to a value.

An expression often assumes the datatype of its components. You can use expressions in many places including:

- Column expressions in a SELECT clause
- A condition of the WHERE clause or HAVING clause

Expressions can be simple or compound. A built-in function such as **length()** or **pi()** can also be considered an expression.

#### Simple Expressions

A simple CCL expression specifies a constant, NULL, or a column. A constant can be a number or a text string. The literal NULL denotes a null value. NULL is never part of another expression, but NULL by itself is an expression.

You can specify a column name by itself or with the name of its stream or window. To specify both the column and the stream or window, use the format "stream\_name.column\_name."

Some valid simple expressions include:

- stocks.volume
- 'this is a string'
- 26

#### Compound Expressions

A compound CCL expression is a combination of simple or compound expressions. Compound expressions can include operators and functions, as well as the simple CCL expressions (constants, columns, or NULL).

You can use parentheses to change the order of precedence of the expression's components.

Some valid compound expressions include:

- sqrt (9) + 1
- ('example' + 'test' + 'string')
- ( length ('example') \*10 ) + pi()

#### Conditional Expressions

A conditional CCL expression evaluates a set of conditions to determine its result. The outcome of a conditional expression is evaluated based on the conditions set. In CCL, the keyword **CASE** appears at the beginning of these expressions and follows a **WHEN-THEN-ELSE** construct.

The basic structure looks like this:

```
CASE
WHEN expression THEN expression
[...]
ELSE expression
END
```

The first **WHEN** expression is evaluated to be either zero or non-zero. Zero means the condition is false, and non-zero indicates that it is true. If the **WHEN** expression is true, the following **THEN** expression is carried out. Conditional expressions are evaluated based on the order specified. If the first expression is false, then the subsequent **WHEN** expression is tested. If none of the **WHEN** expressions are true, the **ELSE** expression is carried out.

A valid conditional expression in CCL is:

```
CASE
WHEN mark>100 THEN grade:=invalid
WHEN mark>49 THEN grade:=pass
ELSE grade:=fail
END
```

#### See also

- HAVING Clause on page 109
- SELECT Clause on page 113
- WHERE Clause on page 116

# **CCL Comments**

Like other programming languages, CCL lets you add comments to document your code.

CCL recognizes two types of comments: doc-comments and regular multi-line comments. Only use doc-comments before CCL top-level statements such as CREATE SCHEMA or CREATE INPUT WINDOW. Doc-comments not immediately preceding a top-level statement are seen as errors by the visual editor with ESP Studio.

Multi-line comments are supported in any context and are therefore recommended over doccomments.

Begin a multi-line comment with / \* and complete it with \* /. For example:

```
/*
This is a multi-line comment.
```

```
All text within the begin and end tags is treated as a comment.
*/
Begin a doc-comment with /** and end it with */. For example:
/**
This is a doc-comment. Note that it begins with two * characters
```

```
instead of one. All text within the begin and end tags is treated
as a comment.
*/
CREATE SCHEMA S1 ...
```

Note that the doc-comment precedes the CREATE SCHEMA statement (provided here as a sample only and not with complete syntax).

It is common to delineate a section of code using a row of asterisks. For example:

CCL treats this rendering as a doc-comment because it begins with /\*\*. To achieve the same effect using a multi line comment, insert a space between the first two asterisks: /\* \*.

# **Case-Sensitivity**

Some CCL syntax elements have case-sensitive names while others do not.

All identifiers are case-sensitive. This includes the names of streams, windows, parameters, variables, schemas, and columns. Keywords are case-insensitive, and cannot be used as identifier names. Adapter properties also include case-sensitivity restrictions.

Most built-in function names (except those that are keywords) and user-defined functions are case-sensitive. While the following built-in function names are case-sensitive, you can express them in two ways:

- setOpcode, setopcode
- getOpcode, getopcode
- setRange, setrange
- setSearch, setsearch
- copyRecord, copyrecord
- deleteIterator, deleteiterator
- getIterator, getiterator
- resetIterator, resetiterator
- businessDay, businessday
- weekendDay, weekendday
- expireCache, expirecache
- insertCache, insertcache
- keyCache, keycache

- getNext, getnext
- getParam, getparam
- dateInt, dateint
- intDate, intdate
- uniqueId, uniqueid
- LeftJoin, leftjoin
- valueInserted, valueinserted

### Example

Two variables, one defined as 'aVariable' and one as 'AVariable' can coexist in the same context as they are treated as different variables. Similarly, you can define different streams or windows using the same name, but with different cases.

### See also

• Appendix A, List of Keywords on page 229

# CHAPTER 5 CCL Query Construction

Use a CCL query to produce a new derived stream or window from one or more other streams/ windows. You can construct a query to filter data, combine two or more queries, join multiple datasources, use pattern matching rules, and aggregate data.

You can use queries only with derived elements, and can attach only one query to a derived element. A CCL query consists of a combination of several clauses that indicate the appropriate information for the derived element. A query is used with the AS clause to specify data for the derived element.

# Filtering

Use the **WHERE** clause in your CCL query to filter data to be processed by the derived elements (streams, windows, or delta streams).

Using the **WHERE** clause and a filter expression, you can filter which incoming data is accepted by your derived elements. The **WHERE** clause restricts the data captured by the **SELECT** clause, reducing the number of results generated. Only data matching the value specified in the **WHERE** clause is sent to your derived elements.

The output of your derived element consists of a subset of records from the input. Each input record is evaluated against the filter expression. If a filter expression evaluates to false (0), the record does not become part of the derived element.

This example creates a new window, IBMTrades, where its rows are any of the result rows from Trades that have the symbol "IBM":

```
CREATE WINDOW IBMTrades
PRIMARY KEY DEDUCED
AS SELECT * FROM Trades WHERE Symbol = 'IBM';
```

#### See also

- Unions on page 36
- Joins on page 37
- Pattern Matching on page 42
- Aggregation on page 43
- WHERE Clause on page 116
- *Expressions* on page 31

# Unions

Use a **UNION** operator in your CCL query to combine the results of two or more queries into a single result.

When combining two or more queries, duplicate rows are eliminated from the result set unless you specify otherwise.

The input for a **UNION** operator comes from one or more streams or windows. Its output is a set of records representing the union of the inputs. This example shows a simple union between two windows, InStocks and InOptions:

```
CREATE INPUT WINDOW InStocks
   SCHEMA StocksSchema
   Primary Key (Ts)
CREATE INPUT WINDOW InOptions
   SCHEMA OptionsSchema
   Primary Key (Ts)
CREATE output Window Union1
   SCHEMA OptionsSchema
   PRIMARY KEY DEDUCED
   AS SELECT s.Ts as Ts, s.Symbol as StockSymbol,
          Null as OptionSymbol, s.Price as Price, s.Volume as
Volume
    FROM InStocks s
UNION
   SELECT s.Ts as Ts, s.StockSymbol as StockSymbol,
           s.OptionSymbol as OptionSymbol, s.Price as Price,
           s.Volume as Volume
   FROM InOptions s
;
```

#### See also

- Filtering on page 35
- Joins on page 37
- Pattern Matching on page 42
- Aggregation on page 43
- UNION Operator on page 114

### **Example: Merging Data from Streams or Windows**

Use the **UNION** clause to merge data from two streams or windows and produce a derived element (stream, window, or delta stream).

1. Create a new window:

CREATE WINDOW name

You can also create a new stream or delta stream.

- 2. Specify the primary key: PRIMARY KEY (...)
- Specify the first derived element in the union: SELECT \* FROM StreamWindow1
- 4. Add the UNION clause: UNION
- **5.** Specify the second derived element in the union:

```
SELECT * FROM StreamWindow2
```

# Joins

Use joins in your CCL query to combine multiple datasources into a single query.

Streams, windows, or delta streams can participate in a join. However, a delta stream can participate in a join only if it has a **KEEP** clause. A join can contain any number of windows and delta streams (with their respective **KEEP** clauses), but only one stream. Self joins are also supported. For example, you can include the same window or delta stream more than once in a join, provided each instance has its own alias.

In a stream-window join the target can be a stream or a window with aggregation. Using a window as a target requires an aggregation because the stream-window join does not have keys and a window requires a key. The **GROUP BY** columns in aggregation automatically forms the key for the target window. This restriction does not apply to delta stream-window joins because use of the **KEEP** clause converts a delta stream into an unnamed window.

Joins are performed in pairs but you can combine multiple joins to produce a complex multitable join. Depending on the complexity and nature of the join, the compiler may create intermediate joins. The comma join syntax supports only inner joins, and the **WHERE** clause in this syntax is optional. When it is omitted, it means that there is a many-many relationship between the streams in the **FROM** clause.

Join Type	Description	
Inner Join	One record from each side of the join is required for the join to produce a record.	
Left Outer Join	A record from the left side (outer side) of the join is produced regardless of whether a record exists on the right side (inner side). When a record on the right side does not exist, any column from the inner side has a NULL value.	

Event Stream Processor supports all join types:

Join Type	Description
Right Outer Join	Reverse of left outer join, where the right side is the outer side and the left side is the inner side of the join.
Full Outer Join	A record is produced whether there is a match on the right side or the left side of the join.

Event Stream Processor also supports these cardinalities:

Туре	Description
One-One	Keys of one side of the join are completely mapped to the keys of the other side of the join. One incoming row produces only one row as output.
One-Many	One record from the one side joins with multiple records on the many side. The one side of the join is the side where all the primary keys are mapped to the other side of the join. Whenever a record comes on the one-side of the join, it produces many rows as the output.
Many-Many	The keys of both side of the join are not completely mapped to the keys of the other side of the join. A row arriving on either side of the join has the potential to produce multiple rows as output.

This example joins two windows (InStocks and InOptions) using the **FROM** clause with ANSI syntax. The result is an output window.

```
CREATE INPUT Window InStocks SCHEMA StocksSchema Primary Key (Ts) ;
```

```
CREATE INPUT Window InOptions SCHEMA OptionsSchema Primary Key (Ts)
KEEP ALL;
```

```
CREATE Output Window OutStockOption SCHEMA OutSchema
   Primary Key (Ts)
   KEEP ALL
AS
   SELECT InStocks.Ts Ts,
       InStocks.Symbol Symbol,
        InStocks.Price StockPrice,
        InStocks.Volume StockVolume,
       InOptions.StockSymbol StockSymbol,
        InOptions.OptionSymbol OptionSymbol,
        InOptions.Price OptionPrice,
        InOptions.Volume OptionVolume
   FROM InStocks JOIN InOptions
     ON
        InStocks.Symbol = InOptions.StockSymbol and
                    InStocks.Ts = InOptions.Ts ;
```

#### See also

• Filtering on page 35

- Unions on page 36
- *Pattern Matching* on page 42
- Aggregation on page 43
- FROM Clause: Comma-Separated Syntax on page 103
- WHERE Clause on page 116
- FROM Clause: ANSI Syntax on page 104
- ON Clause: Join Syntax on page 112
- KEEP Clause on page 91

# **Key Field Rules**

Key field rules ensure that rows are not rejected due to duplicate inserts or the key fields being NULL.

- The key fields of the target are always derived completely from the keys of the many side of the join. In a many-many relationship, the keys are derived from the keys of both sides of the join.
- In a one-one relationship, the keys are derived completely from either side of the relationship.
- In an outer join, the key fields are derived from the outer side of the join. An error is generated if the outer side of the join is not the many-side of a relationship.
- In a full-outer join, the number of key columns and the type of key columns need to be identical in all sources and targets. Also, the key columns require a **FIRSTNONNULL** expression that includes the corresponding key columns in the sources.

When the result of a join is a window, specific rules determine the columns that form the primary key of the target window. In a multitable join, the same rules apply because conceptually each join is produced in pairs, and the result of a join is then joined with another stream or window, and so on.

	One-One	One-Many	Many-One	Many-Many
INNER	Keys from at least one side should be included in the pro- jection list (or a combination of them if keys are composite).	Keys from the right side should be inclu- ded in the projection list.	Keys from the left side should be inclu- ded in the project list.	Keys from both sides should be in- cluded in the projec- tion list.
LEFT	Keys from the left side alone should be included.	Not allowed.	Keys from the left side should be inclu- ded in the projection list.	Not allowed.

This table illustrates this information in the context of join types:

	One-One	One-Many	Many-One	Many-Many
RIGHT	Keys from the right side alone should be included.	Keys from the right side should be inclu- ded in the projection list.	Not allowed.	Not allowed.
OUTER	Keys should be formed using <b>FIRST-</b> <b>NONNULL</b> () on each pair of keys from both sides.	Not allowed.	Not allowed.	Not allowed.

#### See also

- Join Examples: ANSI Syntax on page 40
- Join Example: Comma-Separated Syntax on page 42

# Join Examples: ANSI Syntax

Examples of different join types using the ANSI syntax.

Refer to these inputs for the examples below.

```
CREATE INPUT STREAM S1 SCHEMA (VallS1 integer, Val2S1 integer, Val3S1
string);
CREATE INPUT WINDOW W1 SCHEMA (Key1W1 integer, Key2W1 string, Val1W1
integer, Val2W1 string) PRIMARY KEY (Key1W1, Key2W1);
CREATE INPUT WINDOW W2 SCHEMA (Key1W2 integer, Key2W2 string, Val1W2
integer, Val2W2 string) PRIMARY KEY (Key1W2, Key2W2);
CREATE INPUT WINDOW W2 SCHEMA (Key1W3 integer, Val1W3 integer,
Val2W3 string) PRIMARY KEY (Key1W3);
```

#### Simple Inner Join: One-One

Here, keys can be derived from either W1 or W2.

```
CREATE OUTPUT WINDOW OW1
PRIMARY KEY (Key1W2, Key2W2)
SELECT W1.*, W2.*
FROM W1 INNER JOIN W2 ON W1.Key1W1 = W2.Key1W2 AND W1.Key2W1 =
W1.Key2W2
```

#### Simple Left Join: One-One

The keys are derived from the outer side of the left join. It is incorrect to derive the keys from the inner side because the values could be null.

```
CREATE OUTPUT WINDOW OW1
PRIMARY KEY (Key1W1, Key2W1)
SELECT W1.*, W2.*
FROM W1 LEFT JOIN W2 ON W1.Key1W1 = W2.Key1W2 AND W1.Key2W1 =
W1.Key2W2
```

#### Simple Full Outer Join: One-One

The key columns all have a required FIRSTNONNULL expression in it.

```
CREATE OUTPUT WINDOW OW2

PRIMARY KEY (Key1, Key2)

SELECT FIRSTNONNULL(W1.Key1W1, W2.Key1W2) Key1,

FIRSTNONNULL(W1.Key2W1, W2.Key2W2) Key2, W1.*, W2.*

FROM W1 FULL JOIN W2 ON W1.Key1W1 = W2.Key1W2 AND W1.Key2W1 =

W1.Key2W2
```

#### Simple Left Join: One-Many

All the keys of W2 are mapped and only one key of W1 is mapped in this join. The many-side is W1 and the one-side is W2. The keys must be derived from the many-side.

CREATE OUTPUT WINDOW OW3 PRIMARY KEY (Key1W1, Key2W1) SELECT W1.\*, W2.\* FROM W1 LEFT JOIN W2 ON W1.Key1W1 = W2.Key1W2 AND W1.Val2W1 = W1.Key2W2

#### Simple Inner Join: Many-Many

This is a many-many join because neither of the keys are fully mapped. The keys of the target must be the keys of all the windows participating in the join.

```
CREATE OUTPUT WINDOW OW3
PRIMARY KEY (Key1W1, Key2W1, Key2W1, Key2W2)
SELECT W1.*, W2.*
FROM W1 LEFT JOIN W2 ON W1.Val1W1 = W2.Val11W2 AND W1.Val2W1 =
W1.Val22W2
```

#### Simple Stream-Window Left Join

When a left join involves a stream, the stream must be on the outer side. The target cannot be a window unless it is also performing aggregation.

```
CREATE OUTPUT STREAM OSW1
SELECT S1.*, W2.*
FROM S1 LEFT JOIN W2 ON S1.Key1S1 = W2.Key1W2 AND W1.Val2W1 =
W1.Key2W2
```

#### Complex Join

The keys for OW4 can be derived either from W1 or W2 because of the inner join between the two tables.

```
CREATE OUTPUT WINDOW OW4
PRIMARY KEY DEDUCED
SELECT S1.*, W1.*, W2.*, W3.*
FROM W1 INNER JOIN (W2 LEFT JOIN W3 ON W2.Key1W2 = W3.Key1W3) ON
W1.Key1W1 = W2.Key1W2 AND W1.Key2W1 = W2.Key2W2;
```

#### Complex Stream-Window Join

Here, the join is triggered only when a record arrives on S1. Also, because there is aggregation, the target must be a window instead of being restricted to a stream.

```
CREATE OUTPUT WINDOW OW5

PRIMARY KEY DEDUCED

SELECT S1.* W1.*, W2.*, W3.* //Some column expression.

FROM S1 LEFT JOIN (W1 INNER JOIN (W2 LEFT JOIN W3 ON W2.Key1W2 =

W3.Key1W3) ON W1.Key1W1 = W2.Key1W2 AND W1.Key2W1 = W2.Key2W2) ON

S1.Val1S1 = W1.Key1.Val1

WHERE W2.Key1W2 = 'abcd'

GROUP BY W1.Key1W1, W1.Key2W2

HAVING SUM(W3.Val1W3) > 10;
```

#### See also

- Key Field Rules on page 39
- Join Example: Comma-Separated Syntax on page 42

### Join Example: Comma-Separated Syntax

An example of a complex join using the comma separated syntax.

This join is a complex join of three windows using the comma-separated join syntax. The **WHERE** clause specifies the conditions on which records are joined.

```
CREATE OUTPUT WINDOW OW4
PRIMARY KEY DEDUCED AS
SELECT W1.*, W2.*, W3.*
FROM W1 w1, W2 w2, W3 w3
WHERE w1.Key1W1 = w2.Key1W2 AND w1.Key2W2 = w2.Key2W2 AND w1.Key1W1
= w3.Key1W3;
```

#### See also

- Key Field Rules on page 39
- Join Examples: ANSI Syntax on page 40

# Pattern Matching

Use the **MATCHING** clause in your CCL query to take input from one or more elements (streams, windows, or delta streams) and produce records when a prescribed pattern is found within the input data.

Patterns can check whether or not events occur during a specific time interval, and then send records to downstream streams.

**Attention:** The pattern rule engine matches patterns regardless of the opcode of the input records, unless the opcode is included as part of the pattern matching criteria.

This example creates an output stream, ThreeConsecTrades, which monitors the QTrades streams and sends a new event when it detects three consecutive trades on the same symbol within five seconds. The output of this stream is the symbol of the traded stock, and its latest three prices.

```
CREATE OUTPUT STREAM ThreeConsecTrades
AS
SELECT
T1.Symbol,
T2.Price Price1,
T2.Price Price2,
T3.Price Price3
FROM QTrades T1, QTrades T2, QTrades T3
MATCHING[5 SECONDS: T1, T2, T3]
ON T1.Symbol = T2.Symbol = T3.Symbol
;
```

#### See also

- Filtering on page 35
- Unions on page 36
- Joins on page 37
- Aggregation on page 43
- MATCHING Clause on page 110

# Aggregation

Aggregation allows for input record grouping based on values in the columns set by the **GROUP BY** clause, applies the specified aggregation function such as min, max, sum, count and so forth, and produces one row of output per group.

Records in a group have the same values for the columns specified in the **GROUP BY** clause. The columns specified in the **GROUP BY** clause also needs to be included in the **SELECT** clause because these columns form the key for the target. This is the reason why the primary key for the aggregate window must use the **PRIMARY KEY DEDUCED** clause instead of explicitly specifying a primary key.

In addition to the **GROUP BY** clause, a **GROUP FILTER** and **GROUP ORDER BY** clause can be specified. The **GROUP ORDER BY** clause orders the records in a group by the specified columns before applying the **GROUP FILTER** clause and the aggregation functions. With the records ordered, aggregation functions sensitive to the order of the records such as first, last, and nth can be used meaningfully.

The **GROUP FILTER** clause is executed after the **GROUP ORDER BY** clause and eliminates any rows in the group that do not meet the filter condition. The filter condition that is specified is similar to the one in the **WHERE** clause. The only exception being that a special rank function can be specified. The rank function is used in conjunction with the **GROUP ORDER BY** clause. After the **GROUP ORDER BY** clause is executed every row in the group is ranked from 1 to N.

Now in the **GROUP FILTER** clause one can say rank() < 11, which means that the aggregation function is only applied to the first 10 rows in the group after it has been ordered by the columns specified in the **GROUP ORDER BY** clause.

Finally an optional **HAVING** clause can also be specified. The **HAVING** clause filters records based on the results of applying aggregation functions on the records in a given group. The primary difference is that a **HAVING** clause aggregation operation is allowed and a **WHERE** clause aggregation operation is not.

**Note:** The **GROUP ORDER BY**, **GROUP FILTER**, and **HAVING** clauses can only be specified in conjunction with a **GROUP BY** clause.

#### Example

The following example computes the total number of trades, maximum trade price, and total shares traded for every Symbol. The target window only has those Symbols where the total traded volume is greater than 5000.

```
CREATE INPUT STREAM Trades
SCHEMA (TradeId integer, Symbol string, Price float, Shares integer);
CREATE OUTPUT WINDOW TradeSummary
PRIMARY KEY DEDUCED
AS
SELECT trd.Symbol, count(trd.TradeId) NoOfTrades, max(trd.Price)
MaxPrice, sum(trd.Shares) TotalShares
FROM Trades trd
GROUP BY trd.Symbol
HAVING sum(trd.Shares) > 5000;
```

### See also

- Filtering on page 35
- Unions on page 36
- Joins on page 37
- Pattern Matching on page 42
- GROUP BY Clause on page 106
- GROUP FILTER Clause on page 107
- GROUP ORDER BY Clause on page 108
- HAVING Clause on page 109

# CHAPTER 6 Advanced CCL Programming Techniques

Use advanced CCL techniques to develop sophisticated and complex projects.

Use declare blocks to define variables, constants, SPLASH functions, and custom datatypes.

Create modules to encapsulate reusable code.

Use explicit memory stores to fine tune performance. Use log stores to retain the contents of named windows on disk, to allow for recovery in the event of a failure.

# **Declare Blocks**

Declare blocks allow a model designer to include elements of functional programming, such as variables, parameters, typedefs, and function definitions in CCL data models.

CCL supports global and local declare blocks.

• **Global declare blocks** – accessible to an entire project; however, you can also set individual global declare blocks for each module.

**Note:** Global declare blocks are merged together if more are imported from other CCL files. Only one is possible per project.

• Local declare blocks – declared in CREATE statements, are accessible only in the SELECT clause of the stream or window in which they are declared.

**Note:** The variables and functions defined in a local declare block are only accessible in the **SELECT** clause and anywhere inside the Flex Operator.

CCL variables allow for the storage of values that may change during the execution of the model. Variables are defined in the declare block using the SPLASH syntax. See the *SPLASH Programmers Guide*.

CCL typedefs are user-defined datatypes and can also be used to create an alias for a standard datatype. A long type name can be shortened using typedef. Once a typedef has been defined in the declare block, it can be used instead of the datatype in all SPLASH statements, and throughout the project.

CCL parameters are constants for which you can set the value at the model's runtime. You can use these parameters instead of literal values in a project to allow behavior changes at runtime, such as window retention policies, store sizes, and other similar changes that can be easily modified at runtime without changing the project. You define CCL parameters in a global declare block, and initialize them in a project configuration file. You can also set a default value for the parameter in its declaration, so that initialization at server start-up is optional.

You can create SPLASH functions in a declare block to allow for operations that are more easily handled using a procedural approach. Call these SPLASH functions from stream queries and other functions throughout the project.

### See also

- Flex Operators on page 50
- Typedefs on page 46
- Parameters on page 47
- Variables on page 48
- CREATE FLEX Statement on page 67

# Typedefs

Declares new names for existing datatypes.

#### Syntax

typedef existingdatatypeName newdatatypeName;

#### Components

existingdatatypeName	The original datatype.	
newdatatypeName	The new name for the datatype.	

### Usage

Typedefs allow giving new names for existing datatypes, which can be used to define new variables and parameters, and specify the return type of functions. Typedefs can be declare in declare blocks, UDFs and inside FLEX procedures. The types declared in typedefs must resolve to simple types.

**Note:** For unsupported datatypes, use a typedef in a declare block to create an alias for a supported datatype.

### Example

This example declares euros to be another name for the money(2) datatype:

```
typedef money(2) euros;
```

Once you have defined the euro typedef, you can use:

euros price := 10.80d2;

which is the same as:

money(2) price := 10.80d2;

### See also

• Datatypes on page 23

• Declaring Project Variables, Parameters, Datatypes, and Functions on page 50

### Parameters

Constants that you set during project setup using the server-command name or the project configuration file.

#### Syntax

```
parameter typeName parameterName1 [:= constant_expression]
[,parameterName2 [:= constant_expression],...];
```

#### Components

typeName	The datatype of the declared parameter.	
parameterName	The name of the declared parameter.	
constant_expression	An expression that evaluates to a constant.	

#### Usage

Parameters are defined using the qualifier **parameter**. Optionally, you can specify a default value. The default value is used only if no value is provided for the parameter at server start-up.

Parameters can use only basic datatypes, and must be declared in the global **DECLARE** block of a project or a module. Parameters cannot be declared with complex datatypes. Since parameters are constant, their value cannot be changed in the model.

See the SPLASH Programmers Guide for information on complex datatypes.

#### Parameters at Project Setup

You can define parameters inside the global declare block for a project and inside the global declare block for a module. Project-level parameters can be bound on server start-up. Module-level parameters are bound when the module is loaded.

Parameters can be assigned values at server start-up time by specifying the values on the command line used to start the server or through the project configuration file. You must provide values for any project parameters that do not have a default value. Parameters can only be bound to a new value when a module or project is loaded.

In the parameter declaration, you can specify a default value. The default value is used for the parameter if it is not bound to a new value when the project or module is loaded. If a parameter does not have a default value, it must be bound when the module or project is loaded, or an error occurs.

When a parameter is initialized with an expression, that expression is evaluated only at compile time. The parameter is then assigned the result as its default value.

When supplying values at runtime for a parameter declared as an interval datatype, interval values are specified with the unit notation in CCL and with a bare microsecond value in the project configuration file. See the *Studio Users Guide* for more information on project configurations and parameters in the project configuration file.

#### See also

• Declaring Project Variables, Parameters, Datatypes, and Functions on page 50

### Variables

Variables represent a specific piece of information that may change throughout project execution. Variables are declared using the SPLASH syntax.

#### Syntax

typeName {variableName[:=any\_expression] [, ...]}

### Usage

Variables may be declared within any declare block, SPLASH UDF, or Flex procedures. Multiple variables may be declared on a single line.

The declaration of a variable can also include a optional initial value, which must be a constant expression. Variables without an initial value initialize to NULL.

Variables can be of complex types. However, complex variables can only be used in local declare blocks and declare blocks within a Flex stream.

Variables declared in a local declare block may subsequently be used in **SELECT** clauses, but cause compiler errors when used in **WHERE** clauses.

For SPLASH language details, see SPLASH Programmers Guide.

### Example

This example defines a variable, then uses the variable in both a regular stream and a FLEX stream.

```
declare
INTEGER ThresholdValue := 1000;
end;
//
// Create Schemas
Create Schema TradeSchema(
    Ts bigdatetime,
    Symbol STRING,
    Price MONEY(4),
    Volume INTEGER
);
Create Schema ControlSchema (
    Msg STRING,
    Value INTEGER
```

```
); //
// Input Trade Window
11
CREATE INPUT WINDOW TradeWindow
 SCHEMA TradeSchema
 PRIMARY KEY (Ts);
11
// Input Stream for Control Messages
11
CREATE INPUT STREAM ControlMsg SCHEMA ControlSchema ;
11
// Output window, only has rows that were greater than the
thresholdvalue
// was when the row was received
CREATE Output WINDOW OutTradeWindow
   SCHEMA (Ts bigdatetime, Symbol STRING, Price MONEY(4), Volume
INTEGER)
   PRIMARY KEY (Ts)
as
select *
   from TradeWindow
   where TradeWindow.Volume > ThresholdValue;
11
//Flex Stream to process the control message
CREATE FLEX FlexControlStream
 IN ControlMsq
 OUT OUTPUT WINDOW SimpleOutput
 SCHEMA ( a integer, b string, c integer)
    PRIMARY KEY ( a)
BEGIN
   ON ControlMsg
        // change the value of ThresholdValue
       if ( ControlMsg.Msg = 'set')
{ThresholdValue:=ControlMsg.Value;}
       // The following is being populate so you can see that the
ThresholdValue is being set
        output [a=ControlMsg.Value; b=ControlMsg.Msg;
c=ThresholdValue; |];
   }
    ;
END
;
```

#### See also

- Declare Blocks on page 45
- SELECT Clause on page 113
- Declaring Project Variables, Parameters, Datatypes, and Functions on page 50

# **Declaring Project Variables, Parameters, Datatypes, and Functions**

Declare variables, parameters, typedefs, and functions in both global and local DECLARE blocks.

- 1. Create a global declare block for your project by using the **DECLARE** statement in your main project file.
- **2.** Add parameters, variables, or user-defined SPLASH functions to the global declare block.

Elements defined in this declare block are accessible to any elements in the project that are not inside a module.

- **3.** Create local declare blocks by using the **DECLARE** statement within derived streams, windows, or both..
- 4. Add variables, parameters, or user-defined SPLASH functions to the local declare block.

These elements are accessible only from within the stream, window, or flex operator in which the block is defined.

#### See also

- *Typedefs* on page 46
- Parameters on page 47
- Variables on page 48

# **Flex Operators**

Flex operators provide extensibility to CCL, allowing custom event handlers, written in SPLASH, to produce derived streams or windows.

A flex operator produces derived streams, windows, or delta streams in the same way that a **CREATE** statement produces these elements. However, a **CREATE** statement uses a CCL query to derive a new window from the inputs, whereas a flex operator uses a SPLASH script.

Flex operators make CCL extensible, allowing you to implement event processing logic that would be difficult to implement in a declarative **SELECT** statement. SPLASH gives you process control and provides data structures that can retain state from one event to the next.

All of the features of SPLASH are available for use in a flex operator, including:

Data structures	• Variables
	• EventCache (windows)
	Dictionaries
	Vectors

Control structures	<ul><li>While</li><li>If</li><li>For</li></ul>

A flex operator can take any number of inputs, and they can be any mix of streams, delta streams, or windows. You can write a splash event handler for each input. When an event arrives on that input, the associated SPLASH script or method is invoked.

You need not have a method for every input. Some inputs may merely provide data for use in methods associated with other inputs; for inputs without an associated method, incoming events do not trigger an action, but are accessible to other methods in the same flex operator.

See the SPLASH Programmers Guide for details on writing the flex methods.

### See also

• CREATE FLEX Statement on page 67

# Modularity

A module in Sybase Event Stream Processor offers reusability; it can be loaded and used multiple times in a single project or in many projects.

Modularity means organizing project elements into self-contained, reusable components called modules, which have well-defined inputs and outputs, and allow you to encapsulate data processing procedures that are commonly repeated.

Modules, along with other objects such as import files and the main project, have their own *scope*, which defines the visibility range of variables or definitions. Any variables, objects, or definitions declared in a scope are accessible within that scope only; they are inaccessible to the containing scope, called the parent scope, or to any other outer scope. The parent scope can be a module or the main project. For example, if module A loads module B and the main project loads module A, then module A's scope is the parent scope to module B. Module A's parent scope is the main project.

Modules have explicitly declared inputs and outputs. Inputs to the module are associated with streams or windows in the parent scope, and outputs of the module are exposed to the parent scope using identifiers. When a module is reused, any streams, variables, parameters, or other objects within the module replicate, so that each version of the module exists separately from the other versions.

You can load modules within other modules, so that module A can load module B, which can load module C, and so on. Module dependency loops, however, are invalid. For example, if module A loads module B, which loads A, the CCL compiler generates an error indicating a dependency loop between modules A and B.

The **CREATE MODULE** statement creates a module that can be loaded multiple times in a project, where its inputs and outputs can be bound to different parts of the larger project. The

**LOAD MODULE** statement allows reuse of a defined module one or more times throughout a project. Modularity is particularly useful when used with the**IMPORT** statement, which allows you to use **(LOAD)** modules created in a separate CCL file.

Note: All module-related compilation errors are fatal.

#### See also

- *Module Creation and Usage* on page 52
- CREATE MODULE Statement on page 74
- IMPORT Statement on page 83
- LOAD MODULE Statement on page 84
- IN Clause on page 90
- *OUT Clause* on page 93
- PARAMETERS Clause on page 94
- STORES Clause on page 98
- Example: Parameters in Modules on page 54
- Example: Creating and Using Modules on page 53

# Module Creation and Usage

Use the **CREATE MODULE** statement to create a reusable module, and **LOAD MODULE** to load a previously created module.

When you load a module, you can connect or bind its input streams or windows to streams in the project. A module's outputs can be exposed to its parent's scope and referenced in that scope using the aliases provided in the **LOAD MODULE** statement.

Parameters inside the module are bound to parameters in the parent scope or to constant expressions. Stores within the module are bound to stores in the parent scope. Binding a store within a module to a store outside the module means that any windows using the module store instead use the bound store.

#### See also

- *Modularity* on page 51
- CREATE MODULE Statement on page 74
- IMPORT Statement on page 83
- LOAD MODULE Statement on page 84
- Example: Creating and Using Modules on page 53

# **Example: Creating and Using Modules**

Use basic concepts of modularity to create a module that processes raw stock trade information and outputs a list of trades with a price exceeding 1.00.

1. Create an import file to group your schemas and allow for reuse throughout the project. In this example, the import file is called schemas.ccl and contains:

```
CREATE SCHEMA TradesSchema (
    Id integer,
    TradeTime date,
    Venue string,
    Symbol string,
    Price float,
    Shares integer
);
```

**Note:** You can define schemas directly inside a module or project; however, this example uses an import file to decrease code duplication and increase maintainability of the CCL.

2. In the project, create a module using the **CREATE MODULE** statement, and import the import file (schemas.ccl) using the **IMPORT** statement.

```
CREATE MODULE FilterByPrice IN TradeData OUT FilteredTradeData
BEGIN
IMPORT 'schemas.ccl';
CREATE INPUT STREAM TradeData SCHEMA TradesSchema;
CREATE OUTPUT STREAM FilteredTradeData SCHEMA TradesSchema
AS SELECT * FROM TradeData WHERE TradeData.Price > 1.00;
END;
```

The module's input stream, TradeData, takes in a raw feed from the stock market, and its output stream, FilteredTradeData, provides filtered results. Using the **IMPORT** statement inside the module allows you to use all of the schemas grouped in the schemas.ccl file in the module streams.

**3.** Load the module into your main project using the **LOAD MODULE** statement. This example also shows how to connect the module to a stock market stream:

```
IMPORT 'schemas.ccl';
CREATE INPUT STREAM NYSEData SCHEMA TradesSchema;
LOAD MODULE FilterByPrice AS FilterOver1 IN TradeData = NYSEData
OUT FilteredTradeData = NYSEPriceOver1Data;
```

- The first line of the project file imports schemas.ccl, which allows the use of the same schema as the module.
- The input stream NYSEData represents trade information from the New York Stock Exchange.

- The LOAD MODULE statement loads the module, FilterByPrice, which is identified by the instance name of FilterOver1.
- Binding the module's input stream, TradeData, with the input stream NYSEData allows information to flow from the NYSEData stream into the module.
- The output of the module is exposed to the project (NYSEPriceOverlData).
- To access the output of the module, select the information from the NYSEPriceOverlData stream.

### See also

- Module Creation and Usage on page 52
- CREATE MODULE Statement on page 74
- IMPORT Statement on page 83
- LOAD MODULE Statement on page 84
- *Modularity* on page 51

### **Example: Parameters in Modules**

Develop your understanding of parameter bindings. Create a module that defines a parameter that can be bound to an expression or to another parameter in the parent scope.

The module FilterByPrice filters all incoming trades based on price, and outputs only the trades that have a price greater than the value in the minimumPrice parameter.

minimumPrice can be set when FilterByPrice is loaded, or it can be bound to another parameter within the project so that the value of minimumPrice is set when the project is loaded on the server.

The module definition is:

```
CREATE MODULE FilterByPrice IN TradeData OUT FilteredTradeData

BEGIN

IMPORT 'schemas.ccl';

DECLARE

parameter money(2) minimumPrice := 10.00d2;

END;

CREATE INPUT STREAM TradeData SCHEMA TradesSchema;

CREATE OUTPUT STREAM FilteredTradeData SCHEMA TradesSchema AS

SELECT * FROM TradeData WHERE TradeData.Price > minimumPrice;

END;
```

#### Binding a Parameter to an Expression

In parameter to expression binding, minimumPrice binds to an expression at the time of loading:

```
LOAD MODULE FilterByPrice AS FilterOver20 IN TradeData = NYSEData OUT
FilteredTradeData = NYSEPriceOver20Data PARAMETERS minimumPrice = 20.00d2;
```

In this type of parameter binding, the module outputs stocks only with a price greater than 20.00.

#### Binding a Parameter in the Module to a Parameter in the Parent Scope

In this type of binding, the parameter inside the module binds to a parameter declared in the main project, therefore modifying the value on which trades are filtered at runtime. This is done by creating a parameter within the project's **DECLARE** block, then binding the parameter (minimumPrice) within the module to the new parameter:

```
DECLARE
parameter money(2) minProjectPrice := 15.00d2;
END;
```

```
LOAD MODULE FilterByPrice AS FilterOverMinProjPrice IN TradeData = NYSEData OUT FilteredTradeData = NYSEPriceOverMinProjPrice PARAMETERS minimumPrice = minProjectPrice;
```

If no value is specified for the project's parameter (minProjectPrice) at runtime, then the module filters based on the project parameter's default value of 15.00. However, if minProjectPrice is given a value at runtime, the module filters based on that value.

#### No Parameter Binding

In this example, minimumPrice has a default value in the module definition, therefore no parameter binding is required when loading the module. The module can be loaded as:

```
LOAD MODULE FilterByPrice AS FilterOver10 IN TradeData = NYSEData OUT FilteredTradeData = NYSEPriceOver10Data;
```

Since no binding is provided in the LOAD MODULE statement, the module filters on its default value of 10.00.

### See also

- Modularity on page 51
- Module Creation and Usage on page 52
- PARAMETERS Clause on page 94
- LOAD MODULE Statement on page 84

# **Data Recovery**

A log store allows data recovery inside a window if a server fails or is shut down.

Log stores provide data recovery for a window. Properly specified log stores recover windows elements on failure, and make sure that data gets restored correctly if the server fails and restarts. You can use log stores with windows that have no retention policy; you cannot use log stores with stateless elements.

When using log stores:

#### CHAPTER 6: Advanced CCL Programming Techniques

- Log stores only store window contents.
- Log stores do not directly store intermediate state, such as variables.
- Local Flex stream variables and data structures are not directly stored. However, they may be regenerated from source data if the source data is in persistent storage.
- Log stores do not preserve opcode information. (During periodic log store compaction and checkpointing, only the current window state is preserved. Records are then restored as inserts.)
- Row arrival order is not preserved. In any stream, multiple operations may be collapsed into a single record during log store compaction, changing arrival order. Inter-stream arrival order is not maintained.
- You can define one or more log stores in a project. When using multiple stores make sure you prevent the occurrence of log store loops. A log store loop is created when, for example, Window1 in Logstore1 feeds Window2 in Logstore2, which feeds Window3 in Logstore1. Log store loops cause compilation errors.
- The contents of memory store windows that receive data directly from a log store window are recomputed once the log store window is restored from disk.
- The contents of memory store windows that receive data from a log store window via other memory store windows are also recomputed, once the input window's contents have been recomputed.

**Note:** If a memory store window receives data from a log store window via a stateless element, for example, a delta stream or a stream, its contents are not restored during server recovery.

Log stores are periodically compacted, at which point all data accumulated in the store is checkpointed and multiple operations on the same key are collapsed. After a checkpoint, the store continues appending incoming data rows to the end of the store until the next checkpoint.

**Note:** The recovery of data written to the store, but not yet checkpointed, is available for input windows only. Sybase recommends that when you assign a window to a log store, you also assign all of its input windows to a log store. Otherwise, data written to the window after the last checkpoint is not restored.

Unlike memory stores, log stores do not extend automatically. Use the CCL **maxfilesize** property to specify log store size. The size of a log store is extremely important. Log stores that are too small can cause processing to stop due to overflow. They can also cause significant performance degradation due to frequent cleaning cycles. A log store that is too large can hinder performance due to larger disk and memory requirements.

#### See also

- Stores on page 18
- CREATE MEMORY STORE Statement on page 73
- CREATE LOG STORE Statement on page 71
- Retention on page 15
- *KEEP Clause* on page 91

# Log Store Optimization Techniques

Specify persistence to optimize data models for maximum performance.

- Whenever possible, create a small log store to store static (dimension) data, and one or more larger log stores for dynamic (fact) data.
- If you are using multiple log stores are being used for larger, rapidly changing, dynamic (fact) data, try to organize the stores on different RAID volumes.
- The correct sizing of log stores is extremely important. See *Sizing a Log Store* in the *Administrators Guide*.

# **Error Streams**

Error streams gather errors and the records that caused them.

#### Description

The error stream provides a means to capture error information along with the data that caused the error. This can assist in debugging errors during development. It can also provide real-time monitoring of projects in a production environment.

You can specify more than one error stream in a single project.

An error stream is identical to other user-defined streams, except it:

- Receives records from its source stream or window only when there is an error on the source stream or window. The record it receives is the input to the source stream or window that caused the error.
- Has a predefined schema that cannot be altered by the user.

Column	Datatype	Description
errorCode	integer	The numeric code for the error that was reported
errorRecord	binary	The record that caused the error
errorMessage	string	Plain text message describing the error
errorStreamName	string	The name of the stream on which this error was reported
sourceStream- Name	string	The name of the stream that sent the record that caused the error
errorTime	bigdatetime	The time the error occurred: a microsecond granularity time- stamp

#### Schema

### Error Codes

- GENERIC\_ERROR
- FP\_EXCEPTION
- BADARGS
- DIVIDE\_BY\_ZERO
- OVERFLOW\_ERR
- UNDERFLOW\_ERR
- SYNTAX\_ERR

#### Limitations

The syntax of the error stream provides a mechanism for trapping runtime errors, subject to these limitations:

- Only errors that occur during record computation are captured in error streams. Errors in computations that occur at server start-up, such as evaluation of expressions used to initialize variables and parameters, are not propagated to error streams. Other errors, such as connection errors and noncomputational errors, are not captured in error streams.
- Errors occurring during computations that happen without a triggering record, such as in the ON START TRANS and ON END TRANS blocks of a flex block, propagate an error record where the errorRecord field contains an empty record.
- For the recordDataToRecord built-in, the stream name must be a string literal constant. This limitation is so that a record type of the return value of the built-in can be determined during compilation.
- The triggering record must be retrieved using provided built-ins. No native nested record support is provided to refer to the record directly.
- The triggering record reported is the immediate input for the stream in which the error happened. This may be a user-defined stream or an intermediate stream generated by the compiler. When using the recordDataToString and recordDataToRecord built-ins, the first argument must match the intermediate stream if one has been generated.
- The subscription utility does not automatically decrypt (convert from binary to ASCII) the error record.
- Output adapters do not automatically decrypt (convert from binary to ASCII) the error record.
- Arithmetic and conversion errors occurring in external functions (C and Java) are not handled; such errors are the users responsibility.
- Error streams are not guaranteed to work within the debugger framework.

### **Monitoring Streams for Errors**

Use error streams to monitor other streams for errors and the events that cause them.

#### Process

- 1. Identify the project and the specific streams to monitor.
- **2.** Determine whether to use multiple error streams. Determine the visibility for each error stream.
- 3. Create the error streams in that project.
- **4.** Display some or all of the information from the error streams in the error record, that is, information aggregated or derived from the error records.

#### Examples

In a project that has one input stream and two derived streams, create a locally visible error stream to monitor all three streams using:

```
CREATE ERROR STREAM AllErrors ON InputStream, DerivedStream1, DerivedStream2;
```

To keep a count of the errors according to the error code reported, add:

```
CREATE OUTPUT WINDOW errorHandlerAgg SCHEMA (errorNum integer, cnt
long)
PRIMARY KEY DEDUCED
AS
SELECT e.errorCode AS errorNum, COUNT(*) AS cnt
FROM AllErrors e
GROUP BY e.errorCode
:
```

In a project that has three derived streams, create an externally visible error stream to monitor only the third derived stream (which calculates a volume weighted average price) using:

CREATE OUTPUT ERROR STREAM vwapErrors ON DerivedStream3;

To convert the format of the triggering record from binary to string, add:

```
CREATE OUTPUT vwapMessages SCHEMA (errorNum integer, streamName
string, errorRecord string) AS
SELECT e.errorcode AS errorNum,
e.streamName AS streamName,
recordDataToString(e.sourceStreamName, e.errorRecord) AS
errorRecord
FROM vwapErrors e;
```

To convert the format of the triggering record from binary to record, add:

```
CREATE OUTPUT vwapMessages SCHEMA (errorNum integer, streamName
string, errorRecord string) AS
SELECT e.errorcode AS errorNum,
e.streamName AS streamName,
recordDataToRecord(e.sourceStreamName, e.errorRecord) AS
```

errorRecord FROM vwapErrors e;
# CHAPTER 7 Statements

The CCL statement reference provides syntax, parameter descriptions, usage, and examples.

# **ADAPTER START Statement**

Controls adapter start times.

#### Syntax

```
ADAPTER START
GROUPS {groupName[NOSTART]},[,...]
...
```

### Usage

Adapter groups are created implicitly when their name is used in the **GROUP** clause of the **ATTACH ADAPTER** statement. The order in which each **groupName** appears determines the order in which the adapter groups start. Adapters that are not assigned to one of the ordered groups are placed in a group that starts after all of the ordered groups have started. By default, all output adapters in a group start in parallel, followed by all input adapters in parallel.

The **ADAPTER START** statement is optional. If the statement is absent, all output adapters start in parallel, followed by all input adapters in parallel.

**NOSTART** identifies adapters that should not start automatically with the rest of the adapters. The user can start these adapters using the external XMLRPC interface (esp\_client.exe).

Errors are generated when ADAPTER START:

- References a group that does not exist.
- Does not reference all adapter start groups created with the ATTACH ADAPTER statement.
- References the same group more than once.

### Example

The **ATTACH ADAPTER** statement creates two named adapters groups (RunGroup1, NoRunGroup), each containing one adapter. The **ADAPTER START** statement is executed with instructions to start RunGroup1. The **NOSTART** syntax instructs the project server not to start NoRunGroup.

```
ATTACH INPUT ADAPTER csvInRun
TYPE dsv_in
TO TradeWindow
GROUP RunGroup1
```

### **CHAPTER 7: Statements**

```
PROPERTIES
  blockSize=1,
  dateFormat='%Y/%m/%d %H:%M:%S',
  delimiter=',',
  dir='$ProjectFolder/../data',
  expectStreamNameOpcode=false,
  fieldCount=0,
  file='stock-trades.csv',
  filePattern='*.csv',
  hasHeader=true,
  safeOps=false,
  skipDels=false,
   timestampFormat= '%Y/%m/%d %H:%M:%S';
ATTACH INPUT ADAPTER csvInNoRun
TYPE dsv in
TO TradeWindow
GROUP NoRunGroup
PROPERTIES
  blockSize=1,
  dateFormat='%Y/%m/%d %H:%M:%S',
  delimiter=',',
  dir='$ProjectFolder/../data',
  expectStreamNameOpcode=false,
  fieldCount=0,
  file='stock-trades.csv',
  filePattern='*.csv',
  hasHeader=true,
  safeOps=false,
  skipDels=false,
   timestampFormat= '%Y/%m/%d %H:%M:%S';
```

ADAPTER START GROUPS NORUNGroup NOSTART, RunGroup1;

# See also

• ATTACH ADAPTER Statement on page 62

# **ATTACH ADAPTER Statement**

Attaches an adapter to a stream or window, or assigns an adapter to a group.

# Syntax

```
ATTACH { INPUT | OUTPUT } ADAPTER name
TYPE type
TO streamorwindow
[GROUP groupName]
[PROPERTIES {prop=value} [, ...]];
```

name	Names to the adapter	
type	Specifies the type of the adapter	
streamorwindow	Specifies the stream or window to which you are attaching the adapter	

# Usage

Adapters are defined with an inline definition of the type and the properties that make up the adapter or else via an adapter property set. The **type** is the unique ID assigned to each adapter. You can find each adapter's type in the *Adapters Guide*.

An ATTACH ADAPTER statement cannot appear after an ADAPTER START statement.

There is no statement that creates adapter groups. You can group adapters by providing the groupname in the **GROUP** clause. This grouping is then later used in the **ADAPTER START** statement to start the adapters in the prescribed order. You cannot specify a group without an **ADAPTER START** statement.

An adapter marked as input can be attached only to an input stream or window. An adapter marked as output can be attached to an input or output stream or window. An adapter (either input or output) cannot be attached to a local stream or window. An adapter defined as an input adapter in its cnxml file cannot be attached as an output adapter, and an adapter defined in its cnxml file as an output adapter cannot be attached as an input adapter.

The property name and value pairs that are valid for an **ATTACH ADAPTER** statement are dependent on the adapter type. The property names are case-insensitive. All specifications relating to what properties are required by a particular adapter exist in that adapter's cnxml file, which is stored in the Sybase Event Stream Processor installation folder. This file is used in the validation of properties.

Any adapter property you provide must have its name defined in the adapter's cnxml file, and the values for all properties must match their defined datatypes. If the same property is provided twice, the compiler raises an error.

You can also specify property sets within an **ATTACH ADAPTER** statement. Property sets are reusable sets of properties that are stored in the project configuration file. If you specify a property set, verify that all required properties are set as individual properties. Property sets override individual properties specified within the **ATTACH ADAPTER** statement.

# Example

```
ATTACH INPUT ADAPTER MacysInventory
TYPE dsv_in
TO InventoryInfo
PROPERTIES
dir='C:/Operations/Stock/Inventory/MacysInventory',
```

```
file='inventory.csv',
propertyset '<name>';
```

• ADAPTER START Statement on page 61

# **CREATE DELTA STREAM Statement**

Defines a stateless element that can interpret all operational codes (opcodes): insert, delete and update.

### Syntax

```
CREATE [ LOCAL | OUTPUT ] DELTA STREAM name
[ schema_clause ]
primary_key_clause
[ local-declare-block ]
as_clause
Query;
```

### Components

name	The name of the delta stream being created.
schema_clause	Schema definition for new windows. If no schema clause is speci- fied, it can be derived from the query.
primary_key_clause	Set primary key. See <i>PRIMARY KEY Clause</i> for more information.
local-declare-block	(Optional) A declaration of variables and functions that can be accessed in the query.
as_clause	Introduces query to statement.
Query	A query implemented in a statement. See <i>Queries</i> for more information.

# Usage

A delta stream is a stateless element that can understand all opcodes. A delta stream can be used when a computation, filter, or union must be performed, when a state does not need be maintained.

However, for a filter, a delta stream modifies the opcode it receives. A delta stream typically forwards the opcode it receives. An input record with an insert opcode that satisfies the filter clause has an insert opcode on the output. An input record with an update opcode, where the update meets the criteria but the original record does not, outputs with an insert opcode.

However, if the old record meets the criteria, it outputs with an update opcode. An input record with a delete opcode outputs with a delete opcode, as long as it meets the filter criteria.

**CREATE DELTA STREAM** is used primarily in computations that transform through a simple projection.

### Restrictions

- A delta stream cannot use functions that cannot be repeated, such as **random()**, since the stream must be able to produce the same result when deleting a record as what was previously produced on the insertion of the same record.
- When subscribing to a delta stream, the opcodes the delta stream generates must be treated as safe opcodes. This means that any inserts/updates must be treated as upserts (insert if the record does not exist and update otherwise). Similarly, any deletes must be treated as a delete if they exists, otherwise silently ignore the delete.
- There are no restrictions on the operations that a target node can perform when using a delta stream as an input.
- When the delta stream is defined using a Flex operator, the SPLASH code can output only inserts or deletes. Upserts and updates are not allowed because the delta streams have no state to handle them correctly. To perform an update, issue a delete, followed by an insert.
- The query of a delta stream cannot contain clauses that perform aggregation or joins.

## Examples

This creates a delta stream that computes total cost:

```
CREATE INPUT WINDOW Trades SCHEMA (

TradeId long,

Symbol string,

Price money(4),

Shares integer

)

PRIMARY KEY (TradeId)

;

CREATE DELTA stream TradesWithCost

PRIMARY KEY DEDUCED

AS SELECT

trd.TradeId,

trd.Symbol,

trd.Price,

trd.Shares,

trd.Price * trd.Shares TotalCost

FROM

Trades trd
```

This creates a delta stream that filters out records where total cost is less than 10,000:

```
CREATE DELTA stream LargeTrades
PRIMARY KEY DEDUCED
AS SELECT * FROM TradesWithCost twc WHERE twc.TotalCost >= 10000
;
```

- DECLARE Statement on page 77
- PRIMARY KEY Clause on page 95
- SCHEMA Clause on page 97
- SELECT Clause on page 113
- AS Clause on page 88
- Chapter 9, Queries on page 101

# **CREATE ERROR STREAM Statement**

Create a stream that collects errors and the events that caused them.

# Syntax

CREATE [LOCAL|OUTPUT] ERROR STREAM name ON source [, source ... ]

name is a string that identifies the newly created error stream.

source is a string that identifies a previously defined stream or window.

# Usage

Error streams collect error data from the specified streams. Each error record includes the error code and the input event that caused the error. You can simply display these records for monitoring purposes, or they may trigger more processing logic downstream, just like the records from other streams.

In production environments, error streams are used for real-time monitoring of one or more streams in the project. They are also used in development environments to monitor the input and derived streams when debugging a project.

The visibility of an error stream is, by default, LOCAL. To make the error stream visible to external monitoring tools or devices, you must specify OUTPUT when you create it.

You can define more than one error stream in a single project.

# Examples

To create a single error stream (that is visible externally) to monitor all the streams in a project with one input stream and two derived streams, enter:

CREATE OUTPUT ERROR STREAM AllErrors ON InputStream, DerivedStream1, DerivedStream2

To create separate error streams (both visible only locally) to monitor the input and derived streams in a project with two input streams and three derived streams, enter:

CREATE ERROR STREAM InputErrors ON InputStream1, InputStream2 CREATE ERROR STREAM QueryErrors ON DerivedStream1, DerivedStream2, DerivedStream3

# **CREATE FLEX Statement**

A flex operator takes input from one or more streams/windows and produces a derived stream or window as its output. It allows the use of SPLASH code to specify customizable processing logic.

**Note:** The name of the Flex operator exists only for labeling in Studio and cannot be referred to in queries. Instead, refer to the output element.

#### Syntax

```
CREATE FLEX procedureName
 IN input1 [KEEP keep_spec], ...
    OUT output_element
        BEGIN
            [ DECLARE
                //variable and function declarations
            END;]
            ON input1 {
                //statements
                };
                [EVERY interval{
                 //periodically executing tasks
                };]
                [ON START TRANSACTION {
                    //tasks to be executed
                    //at the start of every transaction
                };]
                [ON END TRANSACTION {
                    //tasks to be executed
                    //at the end of each transaction
                };]
        END;
OUT output_element
output_element:
{[OUTPUT/LOCAL] STREAM name schema_clause
        [OUTPUT/LOCAL] DELTA STREAM name schema_clause | PRIMARY
KEY{column1,column2,...)
        [OUTPUT/LOCAL] WINDOW name schema_clause}
       [PRIMARY KEY(column1,column2,...)][store_clause][keep_clause]
[aging_clause]
    }
```

#### Components

procedureName	The name of the Flex operator being created.
---------------	--

IN input1	Inputs to the Flex operator are declared in the <b>IN</b> clause. The inputs can be streams, delta streams, windows, or outputs of another flex operator.
<b>KEEP</b> keep_spec	The <b>KEEP</b> clause modifies the retention policy of existing input elements that are either delta streams or windows.
<b>OUT</b> output_element	The output of the Flex operator is defined in the <b>OUT</b> clause. A Flex stream can have only one output. The <b>SCHEMA</b> clause is mandatory for all output types.
DECLARE END;	(Optional) The <b>DECLARE</b> block can define vari- ables and functions of all types, including com- plex data types such as records, vectors, diction- aries and event caches.
	See the <i>SPLASH Programmers Guide</i> for addi- tional information.
ON inputl	The <b>ON</b> input clause must be declared for every input of the Flex operator. The SPLASH code specified in this block is executed each time an input record is received. If an input element does not require processing, use an empty <b>ON</b> input clause.
EVERY interval	(Optional) The <b>EVERY</b> interval clause allows you to specify a block of code that is executed every time the interval expires. The interval can be specified explicitly, or specified through an inter- val type parameter.
ON START TRANSACTION and ON END TRANSACTION	(Optional) The SPLASH statements specified in the START/END transaction block are executed at the start/end of each transaction respectively. You can individually specify a <b>START TRANS-</b> <b>ACTION</b> block or <b>END TRANSACTION</b> block, without the other block.

# Usage

The **CREATE FLEX** statement is used to create a Flex operator that accepts any number of input elements and produces one output element. The input elements are previously existing streams, delta streams, and windows defined in the project. If the input element is a delta stream or window, its retention policy can be modified by specifying a **KEEP** clause. The output element is a stream, delta stream, or window with an unique name generated by the Flex

operator. Specification of the **SCHEMA** clause is mandatory for all output element types. Specification of the **PRIMARY KEY** is mandatory for output elements that are delta streams or windows.

The **ON** input clause contains the processing logic for inputs arriving on a particular input element. Specification of the **ON** input clause is mandatory for each input of the Flex operator. The **ON START TRANSACTION** and **ON END TRANSACTION** clauses are optional and contain processing logic that should be executed at the start/end of each transaction respectively. The optional **EVERY** interval clause contains logic that is executed periodically based on a fixed time interval independent of any incoming events.

# Restrictions

- A **KEEP** clause can be specified for the input of a Flex operator if the input element is a window or a delta stream.
- You cannot declare functions in the **ON** input and **EVERY** clauses.
- You can define event cache types only in the local **DECLARE** block associated with the statement.
- A Flex delta stream (a Flex stream for which the output is a delta stream) cannot be used to generate records with update or upsert opcodes. To generate records with these opcodes, use a Flex window instead of a Flex delta stream.
- The SPLASH output statement can be used inside the body of a function defined only in the local declare block of a Flex operator and not in a global declare block or a local declare block of any other element.

# Example

This example computes the average trade price every five seconds.

```
CREATE FLEX ComputeAveragePrice
IN NASDAQ_Trades
OUT OUTPUT WINDOW AverageTradePrice SCHEMA (Symbol string,
AveragePrice money(4) ) PRIMARY KEY(Symbol)
BEGIN
 DECLARE
  typedef [|money(4) TotalPrice; integer NumOfTrades] totalRec t;
  dictionary(string,totalRec_t) averageDictionary;
END;
ON NASDAQ Trades {
 totalRec_t rec := averageDictionary[NASDAQ_Trades.Symbol];
 if( isnull(rec) ) {
  averageDictionary[NASDAQ_Trades.Symbol] :=
  [|TotalPrice = NASDAQ_Trades.Price; NumOfTrades = 1];
  } else {
  // accumulate the total price and number of trades per input record
  averageDictionary[NASDAQ_Trades.Symbol] :=
   [ TotalPrice=rec.TotalPrice + NASDAQ_Trades.Price;
  NumOfTrades=rec.NumOfTrades + 1];
 }
};
EVERY 5 SECONDS {
 totalRec_t rec;
```

```
for (sym in averageDictionary ) {
  rec := averageDictionary[sym];
  output setOpcode([Symbol=sym;|AveragePrice=(rec.TotalPrice/
  rec.NumOfTrades);], upsert);
  }
};
END;
```

• IN Clause on page 90

# **CREATE LIBRARY Statement**

Before using external C/C++ and Java functions, declare them in your CCL project using the **CREATE LIBRARY** statement.

# Syntax

```
CREATE LIBRARY libraryName LANGUAGE {C|JAVA} FROM fileName(
returnType funcName (argType [argName],...);
...);
```

libraryName	The user-specified name of the library.
C, JAVA	Defines the language of the library. The names are case-insensitive.
fileName	For C/C++ functions, the directory of the shared library. You can use a path relative to the current directory.
	For Java functions, the name of the class file without the .class suffix. You can specify it as a string parameter. You can use the -j option when starting the Event Stream Processor Server to provide the locations of the class files.
funcName	The name of the declared function.
returnType, argType	Datatype of the return value of the function and an argument of the function, respectively.
argName	The name of an argument of the function.

# Components

# Usage

Call declared functions using the libraryName.funcName notation.

Use the **IMPORT** statement to import the **CREATE LIBRARY** statement from a different CCL file to your main project.

You can reference only one external library using the **CREATE LIBRARY** statement, but you can reference the external library any number of times in multiple **CREATE LIBRARY** statements.

Libraries are defined, which means you can use them before they have been declared. However, if a global user-defined function uses an external C/C++ or Java function, you must declare the library, specifying the function signature, before the global DECLARE block.

**Note:** C/C++ external library calls support all datatypes, namely boolean, integer, long, float, money(n), date, bigdatetime, binary, and string.

Java external library calls support only integer, long, double, and string datatypes.

Complex types such as dictionaries, vectors, event caches and record types are not supported in external functions.

#### Examples

Create a C/C++ Library

Create a Java Function

```
CREATE LIBRARY MyJavaFunctions LANGUAGE JAVA FROM 'MyClass' (
    integer MyFunc1 (integer, integer, float);
    string MyFunc2(string);
);
```

### See also

• IMPORT Statement on page 83

# **CREATE LOG STORE Statement**

Creates a log store for use by one or more windows. Unlike a memory store (which is the default) a log store persists data to disk so that it can be recovered after a shutdown or failure.

#### Syntax

```
CREATE [DEFAULT] LOG STORE storename
PROPERTIES
filename='filepath'
[sync={ true | false},]
[sweepamount=size,]
[reservepct=size,]
[ckcount=size,]
[maxfilesize=filesize];
```

filename	The absolute or relative path to the folder where log store files should be written. The relative path is preferred.
maxfilesize	The maximum size of the log store file in MB. Default is 8MB.
sync	Specifies whether the persisted data is updated synchronously with every stream being updated. A value of true guarantees that every record acknowledged by the system is persisted at the ex- pense of performance. A value of false improves performance, but it may result in a loss of data that is acknowledged, but not yet persisted. Default is false.
reservepct	The percentage of the log to keep as free space. Default is 20 percent.
sweepamount	The amount of data, in megabytes, that can be cleaned in a single pass. Default is 20 percent of <b>maxfilesize</b> .
ckcount	The maximum number of records written before writing the in- termediate metadata. Default is 10,000.

#### Parameters

#### Components

storename	An identifier that can be referenced in the <b>STORE</b> clause of stateful elements. Must be unique.
filepath	A path to the log store folder, enclosed in single quotes
size	An integer.
filesize	A size in MB.

# Usage

A log store is a disk-optimized store that is persisted on the disk. The state of windows assigned to a log store are restored upon recovery, and the state of memory store windows that receive data from a log store window are recomputed when possible. Log stores are implemented as memory mapped files. The **filename** parameter is required; however, **sync**, **sweepamount**, **reservepct**, and **ckcount** are optional. If these parameters are not specified, the store refers to their default values.

Specify parameters in the **PARAMETERS** clause, in any order.

You cannot specify memory store parameters for log store parameters, or log store parameters for memory parameters.

If **DEFAULT** is specified, the store is the default store for the module or project. The store is used for stateful elements that do not explicitly specify a store with a **STORE** clause. When a store is not defined for the project or module, a default memory store is automatically created for holding the stateful elements. Due to the restrictions on the use of log stores, making a log store the default store for a project is NOT recommended

# Example

```
CREATE LOG STORE myStore
PROPERTIES
filename='myfile',
maxfilesize=16,
sweepamount=4,
ckcount=15000,
reservepct=20,
sync=false;
```

# See also

- CREATE MEMORY STORE Statement on page 73
- Stores on page 18
- Data Recovery on page 55

# **CREATE MEMORY STORE Statement**

Creates a named memory store that one or more windows can be assigned to. Is not required but can be used for performance optimization.

# Syntax

```
CREATE [DEFAULT] MEMORY STORE storename
[PROPERTIES
[INDEXTYPE={'tree'|'hash'},]
[INDEXSIZEHINT=size]]
```

### Parameters

INDEXTYPE	The type of index mechanism for the stored elements. The default is 'tree'. Use tree for binary trees. Binary trees are predictable in use of memory and consistent in speed. Use hash for hash tables, as hash tables are faster, but they often consume more memory.
INDEXSIZEHINT	(Optional) Determines the initial number of elements in the hash table, when using hash. The value is in units of 1024. Setting this higher consumes more memory, but reduces the chances of spikes in latency. Default is 8KB.

storename	An identifier that can be referenced in the <b>STORE</b> clause of stateful elements. Must be unique.
'tree'	Default index mechanism.
'hash'	Alternative index mechanism.

#### Components

### Usage

A memory store holds all the retained records for one or more windows. The data is held in memory and does not persist on the disk. The **INDEXTYPE** parameter is optional, and the store supports 'tree' or 'hash' index types. If you do not specify the index type and size parameters, the store refers to their default values.

Specify parameters in the **PARAMETERS** clause, but this clause is optional for memory stores, since all its parameters are optional. Properties may be specified in any order.

You cannot specify memory stores parameters for log stores, or log store parameters for memory stores.

If you specify **DEFAULT**, the store is the default store for the module or project. The store is used for stateful elements that do not explicitly specify a store with a **STORE** clause. When a store is not defined for the project or module, a default memory store is automatically created for holding the stateful elements.

### Example

```
CREATE DEFAULT MEMORY STORE Storel PROPERTIES INDEXTYPE='hash', INDEXSIZEHINT=16;
```

# See also

- CREATE LOG STORE Statement on page 71
- Stores on page 18
- Data Recovery on page 55

# **CREATE MODULE Statement**

Create a module that contains specific functionality that you can load in a CCL project using the LOAD MODULE statement.

#### Syntax

```
CREATE MODULE moduleName
IN input1 [,...]
OUT output1[,...]
BEGIN
```

```
statements;
END;
```

## Components

moduleName	The name of the module.
input1	The input stream or window.
output1	The output stream or window.

## Usage

All CCL statements are valid in a module except:

- CREATE MODULE
- ATTACH ADAPTER
- ADAPTER START GROUPS

moduleName should be unique across all object names in the scope in which the statement exists. The names in the **IN** and **OUT** clauses must match the names of the streams or windows defined in the **BEGIN-END** block. All streams or windows with input visibility must be listed in the **IN** clause. All streams, windows, and delta streams (including those created by the flex operator), with output visibility must be listed in the **OUT** clause. The compiler generates an error if any input or output objects exist in the module and are not listed in their respective **IN** or **OUT** clause.

While you can use multiple **CREATE** statements within modules, such as the **CREATE WINDOW** and **CREATE STREAM** statements, the **CREATE STORE** statement uses a special syntax that cannot be used outside of a module. The syntax used within a module does not allow you to specify any store properties. The **CREATE STORE** syntax within a module is: CREATE [DEFAULT] {MEMORY|LOG} STORE store1-inmodule;

### Note: All CREATE MODULE statement compilation errors are fatal.

# Restrictions

• You cannot use the CREATE MODULE statement within the module definition.

# Example

This example creates a simple module that filters data based on a column's values:

```
CREATE MODULE filter_module
IN moduleIn
OUT moduleOut
BEGIN
CREATE SCHEMA filter_schema (Value INTEGER);
CREATE INPUT STREAM moduleIn SCHEMA filter_schema;
CREATE OUTPUT STREAM moduleOut SCHEMA filterSchema AS SELECT *
FROM moduleIn WHERE moduleIn.Value > 10;
END;
```

- CREATE LOG STORE Statement on page 71
- CREATE MEMORY STORE Statement on page 73
- IMPORT Statement on page 83
- LOAD MODULE Statement on page 84

# **CREATE SCHEMA Statement**

Defines a named schema that can be referenced later and reused by one or more streams/ windows in the project or module.

# Syntax

```
CREATE SCHEMA name {(columname type [,...])|
INHERITS [FROM] schema_name [,...] [(columname type [,...])]};
```

## Components

name	An identifier that is referenced while defining stateless or stateful elements.
columnname	The unique name of a column.
type	The datatype of the specified column.
schema_name	The name of another schema.

# Usage

The **CREATE SCHEMA** statement defines a named schema that can be referenced by stateful and stateless elements such as streams or windows. You can define the schema as an inline schema definition, or so that it inherits the definition from another schema.

You can extend a schema by setting it up to inherit an existing schema definition and appending more columns. Additional columns you specify are appended to the inherited schema. Otherwise, the inherited schema definition remains an exact replica of the specified named schema. Alternatively, you can extend a schema by inheriting multiple schema definitions.

The concatenation of the schemas is implicit in the specified order. Additional columns are appended. These column names must be unique, otherwise an error is raised.

# Examples

This creates two schemas, symbol\_schema and trade\_schema, which is extended from symbol\_schema:

```
CREATE SCHEMA symbol_schema (Symbol STRING);
CREATE SCHEMA trade_schema INHERITS FROM symbol_schema (Price FLOAT);
```

• Schemas on page 17

# **DECLARE Statement**

DECLARE block statements specify the variables, parameters, typedefs and functions used in a CCL project.

Syntax

```
DECLARE
[declaration;]
...
END;
```

# Usage

CCL declare blocks consist of a **DECLARE** statement and an **END** statement with zero or more declarations between them.

A **DECLARE** block statement can be used to define variables, typedefs, parameters, and functions. The syntax for each of these declarations is:

- Variables use the SPLASH syntax, and you can specify a default value: datatypeName variableName [:=any\_expression] [,...]
- Typedefs declare new names for datatypes: existingdatatypeName newdatatypeName
- Parameters use the qualifier **parameter**, and you can specify a default value: parameter datatypeName parameterName [:=constant\_expression]

Declare blocks can be local or global. When declare blocks are used inside a **CREATE** stream or window statement they become local declare blocks. A local declare block is visible only inside the stream or window with which it is used. When a **DECLARE** block statement is used inside a module or project, it becomes a global declare block. Global declare blocks are visible anywhere within that project or module.

Terminate each declaration in the **DECLARE** block statement with a semicolon.

# Example

This example demonstrates the **DECLARE** block in the global context, meaning it is outside of any **CREATE** command.

```
declare
    integer toggle(integer x) { if ( x%2 = 0) { return 1; } else
    { return 2; } }
end;
CREATE SCHEMA sc1 (k1 integer,k2 string);
CREATE SCHEMA sc2a (k1 integer,k2 string,k3 string, k4 integer);
create schema s1_104(c2 integer, c3 date, c4 float, c5 string, c6
```

```
money );
CREATE INPUT WINDOW iwin1 SCHEMA sc1 primary key(k1);
CREATE INPUT WINDOW iwin2 SCHEMA sc1 primary key(k1);
create input window w1_104 schema s1_104 primary key(c2);
create delta stream ds2_104 primary key deduced as select * from
w1_104;
```

create output window ww\_innerjoin1 schema sc2a primary key (k1,k2)

This example shows the **DECLARE** block local to a stream, meaning it is inside a **CREATE** command (not flex)

```
declare
    integer i1 := 1;
    string s1 := 'ok';
end
    as
    select A.k1,(A.k2 + s1) k2,B.k2 k3, toggle(A.k1) k4
    from iwin1 A join iwin2 B
    on A.k1 = B.k1
;
create flex flex104
    in ds2_104
    out output stream flexos104 schema s1_104
    begin
```

This example shows a **DECLARE** block local to a flex stream.

```
declare
           integer counter := 0;
       end;
       on ds2_104 {
           counter++;
           output ds2_104_stream[ [c2=ds2_104.c2; ]];
       };
       on end transaction {
           if (counter = 4) {
               typeof( flexos104 ) rec;
               rec := flexos104_stream[ [c2=0; ] ];
               rec.c2 := rec.c2 + counter;
               output rec;
               rec := flexos104_stream[ [c2=1; ] ];
               rec.c2 := rec.c2 + counter;
               output rec;
               rec := flexos104_stream[ [c2=2; ] ];
               rec.c2 := rec.c2 + counter;
               output rec;
               rec := flexos104_stream[ [c2=3; ] ];
              rec.c2 := rec.c2 + counter;
               output rec;
```

```
};
end;
```

- CREATE DELTA STREAM Statement on page 64
- PRIMARY KEY Clause on page 95
- SCHEMA Clause on page 97
- SELECT Clause on page 113
- Declare Blocks on page 45

# **CREATE STREAM Statement**

Create either an input stream that receives events from external sources, or a derived stream of events that is the result of a continuous query applied to one or more inputs.

# Syntax

```
CREATE INPUT STREAM name schema_clause
filter-expression-clause
;
CREATE [ LOCAL | OUTPUT ] STREAM name [ schema_clause ]
[ local-declare-block ]
as_clause
;
```

# Components

schema_clause	Specifies the schema. The schema clause is required for input streams, but is optional for local and output streams. If the schema is not specified for local and output streams, it is deduced automatically by the compiler based on the query specification.
filter-expression-clause	Can be specified on an input stream. This clause filters the events before accepting them from the adapter or an outside publisher. Specify columns in the format <i>node name.column name</i> . For an input stream, use the stream name for the node name. For local or output streams, use the name or its alias from the <b>FROM</b> clause.
local-declare-block	Allows variable and function declarations that can be accessed in expres- sions in the query. You cannot define a local-declare-block on an input stream.
as_clause	Introduces a query to a statement.

### Usage

The **CREATE STREAM** statement explicitly creates a stateless element known as a stream, which can be designated as input, output, or local. Input streams include a mandatory schema, and may include an optional filter expression that can remove unneeded data before further processing. Each incoming event is processed, any output is published, and then the stream is ready to process the next event.

Output and local streams have an optional schema. They can contain a local declare block to define variables and functions that can be used in the **SELECT** clause of the query.

# Example

This creates an input stream with a filter:

```
CREATE INPUT STREAM InStr
SCHEMA (Coll INTEGER, Col2 STRING)
WHERE InStr.Col2='abcd';
```

This creates an output stream where the schema is implicitly determined by the **SELECT** clause:

```
CREATE OUTPUT STREAM OutStr as
SELECT InStr.Coll, InStr.Col2
FROM InStr
WHERE InStr.Col1 > 1000;
```

# See also

- SCHEMA Clause on page 97
- WHERE Clause on page 116
- DECLARE Statement on page 77
- SELECT Clause on page 113
- FROM Clause: ANSI Syntax on page 104
- FROM Clause: Comma-Separated Syntax on page 103
- AS Clause on page 88

# **CREATE WINDOW Statement**

Defines a named window that can be referenced and used by one or more downstream operators or, if an output window, can be used to publish results.

# Syntax

```
CREATE INPUT WINDOW name schema_clause
primary_key_clause
[store_clause]
[keep_clause]
;
```

```
CREATE [ LOCAL | OUTPUT ] WINDOW name schema_clause
{ PRIMARY KEY (column1, column2, ...) | PRIMARY KEY DEDUCED }
[store_clause]
[aging_clause]
[keep_clause]
[local-declare-block]
as_clause
;
CREATE INPUT WINDOW Trades SCHEMA (Id long, Symbol string, Price
money(4)) PRIMARY KEY (Id);
```

### Components

name	A name for the window being created.
schema_clause	Required for input windows, but optional for lo- cal and output windows. When the schema clause is not specified for local and output windows, it is automatically deduced by the compiler.
primary_key_clause	Set primary key.
store_clause	(Optional) Specifies the physical mechanism used to store the state of the records. If no clause is specified, project or module defaults apply.
keep_clause	(Optional) Specifies the retention policy for the window. When not specified, the window uses the KEEP ALL retention policy as a default.
aging_clause	(Optional) Specifies the data aging policy. Used only with output or local windows.
local-declare-block	(Optional) Allows variable and function declara- tions that can be accessed in expressions in the query. You cannot define a local-declare-block on an input stream.
as_clause	Introduces a query to a statement.

# Usage

The **SCHEMA** and **PRIMARY KEY** clause is mandatory for an input window. The **SCHEMA** clause is optional for derived windows. If a **SCHEMA** is not defined the compiler implicitly determines it based on the projection list. For derived windows, the primary key may be either deduced or explicitly specified. There are a few exceptions to these rules, which is noted in the appropriate context.

The **CREATE WINDOW** statement can also includes a **STORE** clause to determine how records are stored, and a **KEEP** clause to determine how many records are stored and for how long. The

window can be of type input, output, or local. Local and output windows can include an **AGING** clause that specifies the data aging policy.

# Example

This example creates a local window containing only position records received in the last ten minutes:

```
CREATE WINDOW TradesAge
PRIMARY KEY DEDUCED
KEEP 10 MINUTES
AGES EVERY 5 SECONDS SET AgeColumn 5 TIMES
AS
SELECT Trades.*, 0 AgeColumn FROM Trades;
```

This example creates a local window containing only position records recieved in the last ten minutes. Inclusion of the local declare block lets you determine how old the record is since last update; the maximum age is 25 seconds. It also enables you to keep track of how many records have been processed (including updates and deletes).

```
CREATE WINDOW TradesAge
PRIMARY KEY DEDUCED
KEEP 10 MINUTES
AGES EVERY 5 SECONDS SET AgeColumn 5 TIMES
DECLARE
    long counter := 0;
    long getRecordCount() {
        return ++counter;
     }
END
AS
SELECT Trades.*, getRecordCount() RecordCount, 0 AgeColumn FROM
Trades;
```

# See also

- SCHEMA Clause on page 97
- PRIMARY KEY Clause on page 95
- STORE Clause on page 98
- AGING Clause on page 87
- KEEP Clause on page 91
- User-Defined SPLASH Functions on page 119
- SELECT Clause on page 113
- FROM Clause: ANSI Syntax on page 104
- FROM Clause: Comma-Separated Syntax on page 103
- AS Clause on page 88

# **IMPORT Statement**

Import libraries, parameters, variables, and schema, function, and module definitions from a CCL file into a project, module, or another **IMPORT** file.

## Syntax

IMPORT 'fileName';

### Component

fileName	The absolute or relative path of the CCL text file you are importing.
	The relative path is relative to the file location of the file that contains the <b>IMPORT</b> statement.

# Usage

Only the following CCL statements are valid in an import file. Any other statements in the file generate compiler error messages.

- IMPORT
- CREATE MODULE
- DECLARE
- CREATE SCHEMA
- CREATE LIBRARY

Any definitions used in an import file must be either defined in the file or imported by the file. Once imported, these definitions belong to the scope into which they are imported. You can use these definitions only in statements that follow the **IMPORT** statement.

Import files can be nested within other import files using the **IMPORT** statement. For example, if file A imports file B, and the project imports file A, then the project has access to every definition within A, which includes all of the definitions within B.

Import cycles are not allowed and are detected by the compiler. For example, if file B imports file A, and file A imports file B, the compiler generates an error message indicating that a cyclical dependency exists between files A and B. Importing the same file twice in a single scope is also not allowed, and results in an error message.

**Note:** You cannot successfully compile your project if you cannot compile the import file, or if the **IMPORT** statement attempts to import an invalid file (an improper file format or the file cannot be found).

# Example

This example imports and uses two schemas.

```
//Defines Schemal
//Imported using relative paths
```

```
IMPORT '../schemas/import1.ccl';
//Defines Schema2
//Imported using absolute paths
IMPORT '~/project/schemas/import2.ccl'; [For UNIX-based systems]
IMPORT 'C:/project/schemas/import2.ccl': [For Windows-based systems]
CREATE INPUT STREAM stream1 SCHEMA Schema1;
CREATE INPUT STREAM stream2 SCHEMA Schema2;
```

- CREATE LIBRARY Statement on page 70
- Modularity on page 51
- CREATE MODULE Statement on page 74
- LOAD MODULE Statement on page 84
- DECLARE Statement on page 77

# LOAD MODULE Statement

The LOAD MODULE statement loads a previously created module into the project.

# Syntax

```
LOAD MODULE modulename AS moduleIdentifier
in-clause
out-clause
[parameters-clause]
[stores-clause];
```

# Components

moduleName	The name of the module, which must match the name of the pre- viously created module.
module identifier	The identifier name must be unique within the parent scope.
<b>IN</b> clause	Binds the input streams or windows defined in the module to pre- viously-created streams or windows in the parent scope.
<b>OUT</b> clause	Exposes one or more output streams defined within the module to the parent scope using unique identifiers.
<b>PARAMETERS</b> clause	Binds one or more parameters defined inside the module to an expression at load time, or binds parameters inside the module to another parameter within the main project. If the parameter has a default value defined, then no parameter binding is required.
STORES clause	Binds a store in the module to a store within the parent scope.

# Usage

All streams in a loaded module have local visibility at runtime, meaning they cannot be subscribed to, published from, or queried. When a module is loaded on the server, all of the streams and windows within the module, and the output streams and windows created by exposing outputs to the parent scope, behave as if they have local visibility. Therefore, the streams and windows within a module and the exposed outputs of the module cannot be queried externally or subscribed to.

LOAD MODULE supports:

- IN clause
- OUT clause
- **PARAMETERS** clause
- **STORES** clause

Note: All LOAD MODULE statement compilation errors are fatal.

## Example

This example uses a module that processes raw stock trade information and outputs a list of trades with a price exceeding 1.00. The module loads into the main project using the **LOAD MODULE** statement.

```
CREATE MODULE FilterByPrice IN TradeData OUT FilteredTradeData
BEGIN
        CREATE SCHEMA TradesSchema (
        Id integer,
        TradeTime date,
        Venue string,
        Symbol string,
        Price float,
        Shares integer
);
        CREATE INPUT STREAM TradeData SCHEMA TradesSchema;
        CREATE OUTPUT STREAM FilteredTradeData SCHEMA TradesSchema
        AS SELECT * FROM TradeData WHERE TradeData.Price > 1.00;
END;
CREATE INPUT STREAM NYSEData SCHEMA TradesSchema;
```

LOAD MODULE FilterByPrice AS FilterOver1 IN TradeData = NYSEData OUT FilteredTradeData = NYSEPriceOver1Data;

# See also

- Modularity on page 51
- IN Clause on page 90
- OUT Clause on page 93
- PARAMETERS Clause on page 94

# **CHAPTER 7: Statements**

• STORES Clause on page 98

# CHAPTER 8 Clauses

Syntax for the various clauses used in statements .

# **AGING Clause**

Specifies the data aging policy.

### Syntax

```
AGES EVERY agingTime SET agingField [maxAgingFieldValue TIMES] [FROM agingTimeField]
```

## Components

agingTime	The time period after which the data aging process begins. Specify the period in hours, minutes, seconds, milliseconds, or microseconds. It can also be specified using a combination of units (for example, 3 MINUTES 30 SECONDS). Can also be specified using the interval parameter.
agingField	The field in the record that is incremented by 1 every time the agingTime period elapses and no activity has occurred on the record.
maxAgingFieldValue	(Optional) The maximum value that agingField is incremented to. If not specified, agingField is incremented once. Can also be specified by the interval parameter.
agingTimeField	(Optional) The field containing the start time for the aging process. For example, if the period of time specified in the agingTime column has elapsed, the data aging process begins. If not specified, the internal row time is used. If specified, the field must contain a valid start time.

# Usage

If data records have not been updated or deleted within a predefined period, they are considered to have aged. When a data record ages, notifications are sent as update events to subscribers of the window.

Note: You can only use the AGING clause with windows.

When the predefined time period (agingTime) elapses, an integer field in the record (agingField) is incremented once, or until a predefined maximum value (maxAgingFieldValue) is reached. The start time of the aging process is specified through the (agingTimeField) field in the record.

If the start time is not explicitly specified, the internal row time is used. When the aging process begins, agingField defaults to 0, and it is incremented by 1 whenever the predefined time period elapses. If a record is updated after aging commences, agingField resets to 0 and the process restarts. If a record is deleted, no aging updates are generated.

When insert is received, the count field sets to 0, the insert is passed through, and aging begins.

Aging starts only after the specified inactivity period. If the data ages every five seconds, then the record must remain inactive for five seconds before it starts counting. A record is considered inactive when no updates or deletes have occurred.

When delete is received, aging stops and the delete is passed through. An update of a record resets the counting to 0.

# Example

This example creates an output window named AgingWindow. The age column for the output window updates every 10 seconds 20 times.

```
CREATE OUTPUT WINDOW AgingWindow
SCHEMA (
AgeColumn integer,
Symbol STRING,
Ts bigdatetime )
PRIMARY KEY (Symbol)
AGES EVERY 10 SECONDS SET AgeColumn 20 TIMES
AS
SELECT 1 as AgeColumn,
TradesWindow.Symbol AS Symbol,
TradesWindow.Ts AS Ts
FROM TradesWindow
;
```

# See also

• CREATE WINDOW Statement on page 80

# **AS Clause**

Introduces a CCL query to a derived element.

### Syntax

```
[...]
AS
CCL Query
[...]
```

# Components

AS	The <b>AS</b> clause introduces a CCL query to the rest of the statement.
CCL Query	The body of the CCL query.

# Usage

The **AS** clause is used within derived elements (streams, windows, and delta streams) to provide a CCL query that determines the type of data processed by the derived element. Because of this, the **AS** clause is valid only with derived elements.

See the *Queries* section for information on structuring a query.

# Example

This example shows the **AS** clause being used to specify the information selected by a derived stream.

```
CREATE STREAM win1 SCHEMA ( coll string )
AS
SELECT inputStream.coll
FROM inputStream;
```

# See also

- Chapter 5, CCL Query Construction on page 35
- CREATE STREAM Statement on page 79
- CREATE WINDOW Statement on page 80
- Chapter 9, Queries on page 101

# **CASE Clause**

Conditional processing evaluates set conditions to determine a result.

### Syntax

CASE

```
WHEN condition THEN expression [...] ELSE expression END
```

### Components

condition	An expression that evaluates to either zero or non-zero. A non-
	zero result indicates the condition is true, a result of zero in-
	dicates the condition is false.

expression	The result of the evaluated conditions. This can be any valid
	expression or variable.

# Usage

A CASE clause is order-dependent and contains conditional expressions that require the parameters **WHEN**, **THEN**, and **ELSE**. **WHEN** conditions filter the specific case and narrow down the result through evaluations of whether the conditions set are true or false. If true, following **THEN** expressions are carried out. If false, subsequent **WHEN** conditions are tested.

If all conditions prior to the **ELSE** parameter are false, then the **ELSE** expression is executed. The CASE clause closes with the keyword **END**.

## Example

This example filters weights and specifies a number to each condition set.

```
CASE
WHEN weight<500 THEN 1
WHEN weight>1000 THEN 3
ELSE 2
END
```

# **IN Clause**

Used in the LOAD MODULE statement to bind inputs in the module to inputs in the parent scope.

### Syntax

IN

```
input1-inModule = input1-parentScope [,...]
```

# Components

input1-inModule	The name of the input stream or window defined in the module.
input1-parentScope	The name of the stream or window in the parent scope. Bind the module input stream or window to this stream.

# Usage

The streams or windows in the parent scope can have any visibility type. Schemas between the bound input streams or windows must be compatible. Schemas are compatible if any one of these requirements is met:

• The number and datatypes of the columns match and are in the same order.

- The stream in the parent scope has more columns than the module stream, and the initial column datatypes match and are in the same order. Any additional columns are ignored by the module, and cannot be primary key columns.
- The parent module stream has fewer columns than the module stream, and the initial column datatypes match and are in the same order. Any additional columns inside the module stream are filled with a NULL value. Primary key columns cannot be null.

Note: For each of these requirements, column names need not match.

When associating inputs, a parent level object that does not have a primary key cannot be bound to a module-level object that requires a primary key. For example, a stream cannot be bound to a window.

#### Restrictions

• All input elements in the module must be bound for the IN clause.

#### Example

This example shows the input streams inside the module (modMarketIn1 and modMarketIn2) being bound to their respective streams in the parent scope, marketIn1 and marketIn2.

```
LOAD MODULE filterModule AS filter1
IN modMarketIn1=marketIn1, modMarketIn2=marketIn2
OUT modMarketOut=marketOut;
```

### See also

- Modularity on page 51
- LOAD MODULE Statement on page 84
- OUT Clause on page 93
- PARAMETERS Clause on page 94
- STORES Clause on page 98

# **KEEP Clause**

Specify either a maximum number of records to retain in a window, or a length of time to retain them.

### Syntax

```
KEEP { count_policy | time_policy } | ALL;
```

Components
------------

count_policy	Specify the maximum number of records that will be retained in the window as either a simple maximum nn ROWS or a maximum with some slack nn ROWS SLACK mm. A larger slack value improves performance by reducing the need to delete one row every time a row is inserted. The number of rows, nn, and the slack value, mm, can be either an integer value or an expression.
time_policy	Specify the length of time that records will be retained in the window as described in the <i>Intervals</i> on page 26 topic.
ALL	Specifies that all of the rows received will be retained.

# Usage

The **KEEP** clause defines a retention policy for a Named or Unnamed Window. Window retention policies include time-based policies (the time duration for which a window retains rows) and count-based policies (the maximum number of rows that the window can retain). If you omit the **KEEP** clause from a window definition, the default policy is KEEP ALL.

When specifying a count-based retention policy, you can specify a **SLACK** value to enhance performance by requiring less frequent cleaning of memory stores. Slack cannot be specified for windows using time-based retention policies.

The location of the **KEEP** clause in the **CREATE WINDOW** statement determines whether a named or an unnamed window is created. When the **KEEP** clause is specified for the window being created, a Named Window is created. If there is a **KEEP** clause in the query portion of the statement, however, an Unnamed Window is implicitly created. This includes cases where there is a **KEEP** clause attached to the **FROM** clause of the query.

# Examples

In the following example, the first **KEEP** clause specifies the retention policy for the named window RecentAvgVols, keeping records for ten minutes. The second instance of the **KEEP** clause creates an unnamed window that stores the average volume over the last five minutes.

```
CREATE WINDOW RecentAvgVols PRIMARY KEY DEDUCED
KEEP 10 MINS
AS
SELECT Trades.Symbol, Avg(Trades.Volume) as Vol
FROM Trades KEEP 5 MINS
GROUP BY Trades.Symbol;
```

This example creates a Sliding Window that retains the most recent 100 records that match the filter condition. The SLACK value means the window may contain as many as 110 records.

```
CREATE WINDOW Last100Trades PRIMARY KEY DEDUCED KEEP 100 ROWS SLACK 10 \,
```

```
AS SELECT * FROM Trades
WHERE Trades.Volume > 1000;
```

- Joins on page 37
- CREATE DELTA STREAM Statement on page 64
- CREATE WINDOW Statement on page 80
- *Retention* on page 15
- Data Recovery on page 55

# **OUT Clause**

Used in the LOAD MODULE statement to expose outputs in the module to the parent scope.

### Syntax

#### OUT

output1-inModule = output1-parentScope [,...]

#### Components

output1-inModule	The name of the output defined in the module.
output1-parentScope	The name by which the output is exposed to the parent scope.

### Usage

The exposed output stream created by the **LOAD MODULE** statement has local visibility, meaning that you cannot attach an output adapter directly to the output stream directly. Outputs are exposed to the parent scope using the outputl-parentScope identifier. The output mapping provides a unique name for the module output so that it can be referred to in the parent scope.

### Restrictions

• At least one output stream must be exposed to the parent scope.

### Example

This example exposes the outputs of the module, modFilteredOut and marketAverageOut, using the respective names filteredOut and averageOut.

```
LOAD MODULE filterModule AS filter1
IN modMarketIn=marketIn1
OUT modFilteredOut=filteredOut, marketAverageOut=averageOut;
```

- Modularity on page 51
- LOAD MODULE Statement on page 84
- IN Clause on page 90
- PARAMETERS Clause on page 94
- STORES Clause on page 98

# **PARAMETERS** Clause

Used in the **LOAD MODULE** statement to provide the bindings for the parameter inside the module at load time.

# Syntax

PARAMETERS

```
parameter1-inModule = value-parentScope [,...]
```

# Components

parameter1-inModule	The name of the parameter defined in the module.
value-parentScope	The value in the parent scope being bound to. This value can be an expression or another parameter defined in the parent scope.

# Usage

Binding a parameter refers to the process of providing a value for a parameter within the module at load time. This means that you can provide a value for the parameter that is specific to each instance of the module. In the **LOAD MODULE** statement, you can bind a parameter inside the module to:

- Another parameter declared within the parent scope, or,
- An expression when loading the module.

**Note:** Expressions involving parameters or variables are evaluated at compile time using the parameter's default value and the variable's initial value. A parameter or variable in a binding expression without a default value generates an error.

You cannot directly bind a parameter defined within a module at runtime; doing so generates a server warning. You can bind module parameters using only the LOAD MODULE statement.

# Example

This example maps the parameters in the module to another value (minValue=2) and to another parameter (maxValue=serverMaxValue).

```
CREATE MODULE filterModule
IN filterIn
```

```
OUT filterOut
BEGIN
   CREATE SCHEMA filterSchema (Value Integer);
   DECLARE
        PARAMETER Integer minValue := 4i
        PARAMETER Integer maxValue;
    END;
    CREATE INPUT STREAM filterIn SCHEMA filterSchema;
    CREATE OUTPUT STREAM filterOut SCHEMA filterSchema AS SELECT *
FROM filterIn WHERE filterIn.Value > minValue and filterIn.Value <
maxValue;
END;
DECLARE
    PARAMETER Integer serverMaxValue;
END;
LOAD MODULE filterModule AS filter1
IN filterIn=marketIn
OUT filterOut=marketOut
PARAMETERS minValue=2, maxValue=serverMaxValue;
```

- Modularity on page 51
- LOAD MODULE Statement on page 84
- IN Clause on page 90
- *OUT Clause* on page 93
- STORES Clause on page 98
- Example: Parameters in Modules on page 54

# **PRIMARY KEY Clause**

Specifies the primary key for a delta stream or window.

# Syntax

```
PRIMARY KEY (column [,...]) | PRIMARY KEY DEDUCED
```

# Components

column	The name of a column in the element's schema
--------	--

# Usage

A primary key uniquely identifies a record, and is required for windows and delta streams.

The primary key is normally treated as "strict." Any records that violate consistency rules, such as an insert of an existing record, or update or delete for a nonexistent record, are discarded and reported in the log.

# CHAPTER 8: Clauses

The primary key is treated as "lax" when a keep policy is placed on a window. The expiration of records caused by the **KEEP** clause creates inconsistencies with incoming records. An insert on an existing record is treated as an update, and an update on a nonexistent record is treated as an insert. A delete on a nonexistent record is silently ignored (as safedelete). This behavior manifests when two records in a chain have expiry policies, and it is apparent that the target window has a smaller expiry period.

# Usage: Explicit Primary Key

An explicitly defined primary key uses the **PRIMARY KEY** clause and refers to one or more columns of the window or delta stream's schema. When a primary key is specified, the engine enforces the constraint, and erroneous operations are flagged as bad records and discarded at runtime. To avoid this issue, ensure the primary key is defined correctly.

# Usage: Deduced Primary Key

If the primary key is specified as **PRIMARY KEY DEDUCED**, the compiler automatically deduces the primary key. If the primary key cannot be deduced, a compilation error is generated.

The primary key is deduced as follows:

- Primary keys cannot be deduced for input windows and Flex operators. They need to be explicitly specified.
- For single source queries, except aggregations, the primary key is deduced from the source. All the key columns from the source needs to copied verbatim for the key deduction to succeed.
- For aggregation the primary keys are the columns in the projection containing the group by expressions.

**Note:** All **GROUP BY** clauses needs to be included in the projection list. If the same expression appears in more than one column then the first column with the **GROUP BY** clause is made the primary key.

For joins, the following rules apply:

- For a left outer join and right outer join the keys are derived from the outer side. For example, the left side in the case of a left join and the right side in the case of a right join. All key columns from the outer side must be present in the projection for the primary key deduction to work correctly.
- For a inner join it depends on the cardinality of the join. For a one-many cardinality the key is derived from the many side. For a many-many cardinality the deduced key is combination of the keys from both sides of a join. For a one-one the key is deduced from one of the sides. The side that is chosen as a key cannot be reliably determined. In all cases the candidate key columns must be copied from the sources directly for key deduction to work correctly.
- For a full outer join the columns containing only a coalesce() function with the key fields of both sides of the join as arguments is deduced to be the key column.
· For the joins of multiple windows, these rules are applied transitively

### See also

- CREATE DELTA STREAM Statement on page 64
- DECLARE Statement on page 77
- SCHEMA Clause on page 97
- SELECT Clause on page 113
- CREATE WINDOW Statement on page 80

# **SCHEMA Clause**

Provides a schema definition for new streams and windows.

#### Syntax

SCHEMA name | (column type [,...])

#### Components

name	The name of schema previously defined with a <b>CREATE SCHEMA</b> statement.
column	The name of a column.
type	The datatype of the column's entries.

#### Usage

A **SCHEMA** clause defines the columns and datatypes (inline schema) in a stream or window, or refers to a previously defined named schema. It may also refer to a schema imported from a different CCL file.

The schema clause is mandatory for input streams, input windows and Flex operators. For all other cases it is optional. In which case the schema is implicitly determined by the columns in the projection list.

In the case of **UNION**, if a schema is not explicitly specified then it is implicitly determine from the first **SELECT** statement in the **UNION**.

- CREATE DELTA STREAM Statement on page 64
- DECLARE Statement on page 77
- PRIMARY KEY Clause on page 95
- SELECT Clause on page 113
- CREATE STREAM Statement on page 79
- CREATE WINDOW Statement on page 80

• UNION Operator on page 114

# **STORE Clause**

Assigns the store for the window in any window definition.

## Syntax

STORE storename

## See also

- Data Recovery on page 55
- CREATE WINDOW Statement on page 80

# **STORES Clause**

Used in the LOAD MODULE statement to bind stores in the module to stores in the parent scope.

## Syntax

```
STORES
store1-inModule = store1-parentScope [,...]
```

## Components

store-inModule	The name of the store defined in the module.
storel-parentScope	The name of the store in the parent scope. Bind the module store to this store.

## Usage

Unbound stores generate compilation errors. When you create windows without specifying a store, and do not create a default store, a default parser-generated memory store is temporarily created for the module. When you load the module, this parser-generated store is assigned to the default memory store of the parent scope. If no default memory store exists in the parent scope, the parser-generated memory store in the module is assigned to a parser-generated memory store created in the parent scope.

**Note:** Modules can participate in store dependency loops. Since all dependency loops are invalid, the instance of a dependency loop within a module will render the project unable to compile.

#### Restrictions

• You can bind stores only of the same type. For example, bind a log store with another log store, and a memory store with another memory store.

#### Example

This example maps a store in the module to a store in its parent scope.

```
CREATE MODULE filterModule
IN filterIn
OUT filterOut
BEGIN
    CREATE MEMORY STORE filterStore;
    CREATE SCHEMA filterSchema (ID Integer, Value Integer);
   CREATE INPUT WINDOW filterIn SCHEMA filterSchema PRIMARY KEY ID
STORE filterStore;
   CREATE OUTPUT WINDOW filterOut SCHEMA filterSchema PRIMARY KEY
DEDUCED STORE filterSTore AS SELECT * FROM filterIn WHERE
filterIn.Value > 10;
END;
CREATE MEMORY STORE mainStore;
CREATE SCHEMA filterSchema (ID Integer, Value Integer);
LOAD MODULE filterModule AS filter1
IN filterIn=marketIn
OUT filterOUT=marketOut
STORES filterStore=mainStore;
```

- Modularity on page 51
- LOAD MODULE Statement on page 84
- IN Clause on page 90
- OUT Clause on page 93
- PARAMETERS Clause on page 94

## **CHAPTER 8: Clauses**

# CHAPTER 9 Queries

Build a query using clauses and operators to specify its function. This section provides reference for queries, query clauses, and operators.

#### Syntax

select\_clause
from\_clause
[matching\_clause]
[where\_clause]
[groupFilter\_clause]
[groupBy\_clause]
[groupOrder\_clause]
[having\_clause]

#### Components

select_clause	Defines the set of columns to be included in the output. See below and <i>SELECT Clause</i> for more information.
from_clause	Selects the source data is derived from. See below and <i>FROM Clause</i> for more information.
matching_clause	Used for pattern matching. See <i>MATCHING</i> <i>Clause</i> and <i>Pattern Matching</i> for more informa- tion.
where_clause	Performs a filter. See <i>WHERE Clause</i> and <i>Filters</i> for more information.
groupFilter_clause	Filters incoming data in aggregation. See <i>GROUP FILTER Clause</i> and <i>Aggregation</i> for more information.
groupBy_clause	Specifies what collection of rows to use the ag- gregation operation on. See <i>GROUP BY Clause</i> and <i>Aggregation</i> for more information.
groupOrder_clause	Orders the data in a group before aggregation. See <i>GROUP ORDER BY Clause</i> and <i>Aggregation</i> for more information.

 Filters data that is output by the derived compo- nents in aggregation. See <i>HAVING Clause</i> and
Aggregation for more information.

## Usage

Queries can use the aforementioned clauses to fulfill various functions, as outlined in the *CCL Query Constructions* chapter. However, the basic structure remains the same when starting a query. The example below demonstrates the use of both the **SELECT** clause and **FROM** clause as would be seen in any query.

The **SELECT** clause is used directly after the **AS** clause. The purpose of the **SELECT** clause is to determine which columns from the source or expressions the query is to use.

Following the **SELECT** clause, the **FROM** clause names the source used by the query. Following the **FROM** clause, implement available clauses to use filters, unions, joins, pattern matching, and aggregation on the queried data.

## Example

This example obtains the total trades, volume, and VWAP per trading symbol in five minute intervals.

```
[...]
SELECT
    q.Symbol,
    (trunc(q.TradeTime) + (((q.TradeTime - trunc(q.TradeTime))/
300)*300)) FiveMinuteBucket,
    sum(q.Shares * q.Price)/sum(q.Shares) Vwap,
    count(*) TotalTrades,
    sum(q.Shares) TotalVolume
FROM
    QTrades q
[...]
```

- Chapter 5, CCL Query Construction on page 35
- FROM Clause: Comma-Separated Syntax on page 103
- FROM Clause: ANSI Syntax on page 104
- GROUP BY Clause on page 106
- GROUP FILTER Clause on page 107
- GROUP ORDER BY Clause on page 108
- HAVING Clause on page 109
- MATCHING Clause on page 110
- ON Clause: Join Syntax on page 112
- SELECT Clause on page 113
- UNION Operator on page 114
- WHERE Clause on page 116

• AS Clause on page 88

# **FROM Clause**

Identifies the stream(s) or window(s) or both that will provide the input to the query.

## See also

- FROM Clause: Comma-Separated Syntax on page 103
- FROM Clause: ANSI Syntax on page 104

## FROM Clause: Comma-Separated Syntax

Specify a single input to a query or use to list tow or more inputs in a join or for pattern matching two data sources in a query, in combination with the **WHERE** clause, using an alternative comma-separated syntax.

## Syntax

```
FROM { stream [ [AS] alias ] | stream [ [AS] alias] keep_clause |
window_name [ [AS] alias] } [, ...]
```

## Components

stream	The name of a data stream
alias	An alias for the stream or window
keep_clause	The policy that specifies how rows are maintained in the window
window_name	The name of a window

## Usage

Use the **FROM** clause with comma-separated syntax for single-source queries, inner joins, and queries that use the **MATCHING** clause. This syntax specifies one or more data sources in a query. Any column or datasource references in the query's other clauses must be to one of the data sources named in this clause.

The comma-separated **FROM** clause can contain multiple data sources connected with an inner join. The multiple sources are separated by commas. The **WHERE** clause, required when using comma-separated syntax, creates the selection condition for the join.

Use comma-separated syntax for the **FROM** clause with a **MATCHING** clause to specify data sources that should be monitored for a specified pattern. The list of data sources can include only data streams, must include all data sources specified in the **MATCHING** clause, and cannot include any other data source.

Use aliases to abbreviate stream or window names, and if required, for differentiating between instances when the same data stream or window is used more than once in the **FROM** clause.

#### See also

- Key Field Rules on page 39
- Joins on page 37
- Join Example: Comma-Separated Syntax on page 42
- FROM Clause on page 103

## FROM Clause: ANSI Syntax

Joins two data sources in a query using outer or inner join syntax.

#### Syntax

```
FROM { stream [ [AS] alias] | stream [ [AS] alias] keep_clause |
window_name [ [AS] alias] | nested_join }
[ RIGHT | LEFT | FULL ] JOIN
{ stream [ [AS] alias] | stream [ [AS] alias] keep_clause |
window_name [ [AS] alias] | nested_join }
nested_join
```

#### Components

stream	The name of a data stream
alias	An alias for the stream or window
keep_clause	The policy that specifies how rows are maintained in the window
window_name	The name of a window
nested_join	A nested join — see below

#### nested\_join

```
FROM { stream [ [AS] alias] | stream [ [AS] alias] keep_clause |
window_name [ [AS] alias] | nested_join }
[ RIGHT | LEFT | FULL ] JOIN
{ stream [ [AS] alias] | stream [ [AS] alias] keep_clause |
window_name [ [AS] alias] | nested_join }
[ on_clause ]
```

#### Components

stream	The name of a data stream
alias	An alias for the stream or window
keep_clause	The policy that specifies how rows are maintained in the window
window_name	The name of a window

nested_join	A nested join. See Joins for more information
on_clause	The join condition

## Usage

For outer joins, use an **ON** clause to specify the join condition. This is optional for inner joins.

You can use this variation of <b>FROM</b> to	create inner, left, right, and full joins:
--	--

JOIN	All possible combinations of rows from the intersection of both data sources (limited by the selection condition, if one is specified) are published.
RIGHT JOIN	All possible combinations of rows from the intersection of both data sources (limited by the selection condition, if one is specified) are published. All the other rows from the right data source are also published. Unmatched columns in the left data source publish a value of NULL .
LEFT JOIN	All possible combinations of rows from the intersection of both data sources (limited by the selection condition, if one is specified) are published. All the other rows from the left data source are also published. Unmatched columns in the right data source publish a value of NULL.
FULL JOIN	All possible combinations of rows from the intersection of both data sources (limited by the selection condition, if one is specified) are published. All other rows from both data sources are published as well. Unmatched columns in either data source publish a value of NULL .

The data sources used with this syntax can include data stream expressions, named and unnamed window expressions, and queries. You can use aliases for datasources in this variation of the **FROM** clause.

The join variation of the **FROM** clause (ANSI syntax) is limited to two datasources. Accommodate additional datasources using a nested join as one of the datasources. If a nested join is used, it can optionally be enclosed in parentheses, and can include its own **ON** clause. The rules for the use of the **ON** clause with a nested join are the same as the rules that govern the use of the **ON** clause in the join containing the nested join.

## Restrictions

- Any column or datasource references in the query's other clauses must be to one of the data sources named in this clause.
- For a left outer join, the data stream can only be on the left side. For a right outer join as well, the data stream can only be on the right side.
- A full outer join cannot join a window to a data stream.

## See also

• Key Field Rules on page 39

- Joins on page 37
- Join Examples: ANSI Syntax on page 40
- FROM Clause on page 103

# **GROUP BY Clause**

Specifies the expressions on which to perform an aggregation operation.

#### Syntax

```
GROUP BY expression1 [, expression2 ...]
```

#### Components

expression	An expression using constants, which can contain
	one or more expressions from the input window
	or stream. However, an expression cannot use
	aggregate functions.

## Usage

It combines one or more result rows into a single row of output. A **GROUP BY** clause is used when the query result contains aggregate functions to specify what expressions to perform the aggregation operation on.

When a **GROUP BY** clause is used in a query, the associated window needs to have a primary key that is deduced by the compiler. If more than one column has the same expression, the first column is used if it has not already been matched with a **GROUP BY** expression.

Note: Every expression in the GROUP BY clause must also be in at least one SELECT column expression.

Use the actual expression in the **GROUP BY** clause, rather than the alias for that expression which prevents the project from compiling. For example, use the expression T.Symbol instead of an alias like Symbol.

## Example

The **GROUP BY** clause collects together the rows according to T.Symbol:

```
CREATE WINDOW Window1 SCHEMA (Symbol STRING, MaxPrice INTEGER)

PRIMARY KEY DEDUCED

KEEP ALL

AS

SELECT T.Symbol, max(T.Price) MaxPrice

FROM Trades T

GROUP FILTER rank() < 10

GROUP BY T.Symbol

GROUP ORDER BY T.Volume DESC

HAVING max(T.Price) > 100 AND T.Symbol ='IBM';
```

## See also

- Aggregate Functions on page 126
- Aggregation on page 43
- GROUP FILTER Clause on page 107
- GROUP ORDER BY Clause on page 108

# **GROUP FILTER Clause**

Filters data in a group before the aggregation operation is performed.

#### Syntax

GROUP FILTER expression

#### Components

expression	Any Boolean expression that does not use aggre- gate functions such as <b>min()</b> or <b>max()</b> . The ex- pression may use columns from the source
	streams or windows.

#### Usage

The **GROUP FILTER** clause filters data before the aggregation operations are applied to the rows. The **GROUP FILTER** clause is used with the **GROUP BY** clause. If **GROUP FILTER** is used with the **GROUP ORDER BY** clause, **GROUP ORDER BY** is executed before **GROUP FILTER**.

The expression in the **GROUP FILTER** clause often uses filters based on functions such as **rank()**. These functions restrict rows that are used in the aggregation. The **rank()** function assigns a rank to each of the individual records in a group. **rank()** is meaningful only when used with the **GROUP ORDER BY** clause.

## Example

The **GROUP FILTER** clause filters out the chosen rows, keeping only those with a rank of less than 10:

```
CREATE WINDOW Window1 SCHEMA (Symbol STRING, MaxPrice INTEGER)

PRIMARY KEY DEDUCED

KEEP ALL

AS

SELECT T.Symbol, max(T.Price) MaxPrice

FROM Trades T

GROUP FILTER rank() < 10

GROUP BY T.Symbol

GROUP ORDER BY T.Volume DESC

HAVING max(T.Price) > 100 AND T.Symbol ='IBM';
```

#### See also

- Aggregate Functions on page 126
- Aggregation on page 43
- GROUP BY Clause on page 106
- GROUP ORDER BY Clause on page 108

# **GROUP ORDER BY Clause**

Orders the data in a group before applying the **GROUP FILTER** clause and aggregating the data.

#### Syntax

```
GROUP ORDER BY column [ASC[ENDING] DESC[ENDING]] [, ...]
```

#### Components

column	Any column in the source streams or windows.
	You can order by more than one column.

#### Usage

The **GROUP ORDER BY** clause is used with the **GROUP BY** clause. Rows may be ordered by one or more columns in the stream or window. **GROUP ORDER BY** orders the data in a group before applying aggregation operations (and before applying **GROUP FILTER**).

Use ASC and DESC keywords to organize column data in ascending or descending order. If no keyword is specified, the default is ascending order.

When used with a **GROUP FILTER** clause, **GROUP ORDER BY** is performed before **GROUP FILTER**. The **GROUP ORDER BY** clause orders records in each group based on the ordering criteria specified in the clause.

## Example

The GROUP ORDER BY clause organizes the chosen rows by T.Volume in descending order:

```
CREATE WINDOW Window1 SCHEMA (Symbol STRING, MaxPrice INTEGER)

PRIMARY KEY DEDUCED

KEEP ALL

AS

SELECT T.Symbol, max(T.Price) MaxPrice

FROM Trades T

GROUP FILTER rank() < 10

GROUP BY T.Symbol

GROUP ORDER BY T.Volume DESC

HAVING max(T.Price) > 100 AND T.Symbol ='IBM';
```

## See also

• Aggregate Functions on page 126

- *Aggregation* on page 43
- GROUP FILTER Clause on page 107
- GROUP BY Clause on page 106

# **HAVING Clause**

Filters rows that have been grouped by a grouping clause.

#### Syntax

HAVING expression

#### Components

expression	Any Boolean expression. Can include aggregate
	functions, as well as simple filters on columns.

#### Usage

The **HAVING** clause is semantically similar to the **WHERE** clause, but can be used only in a query that specifies a **GROUP BY** clause. The **HAVING** clause filters rows after they have been processed by the **GROUP BY** clause. Unlike the **WHERE** clause, the **HAVING** clause allows the use of aggregates in the expression. Its function is to eliminate some of the grouped result rows.

#### Example

The **HAVING** clause filters the rows that have been grouped by the **GROUP FILTER**, **GROUP BY**, and **GROUP ORDER** clauses:

```
CREATE WINDOW Window1 SCHEMA (Symbol STRING, MaxPrice INTEGER)

PRIMARY KEY DEDUCED

KEEP ALL

AS

SELECT T.Symbol, max(T.Price) MaxPrice

FROM Trades T

GROUP FILTER rank() < 10

GROUP BY T.Symbol

GROUP ORDER BY T.Volume DESC

HAVING max(T.Price) > 100 AND T.Symbol ='IBM';
```

- Aggregate Functions on page 126
- Aggregation on page 43
- GROUP BY Clause on page 106
- GROUP FILTER Clause on page 107
- GROUP ORDER BY Clause on page 108
- *Expressions* on page 31

# **MATCHING Clause**

This is used within a query for pattern matching, which allows detection of patterns of events across one or more sources.

Note: This form of the ON clause is different from the ON clause with JOIN syntax. You cannot specify both forms at the same time.

#### Syntax

. .

#### Components

MATCHING	Identifies the MATCHING clause.
interval:pattern	interval specifies the interval and pattern specifies the matching patterns.
source.column	The name of the source input and the column.
getOpcode()	Includes opcode conditions on the pattern.
opcode_constant	Specifies the opcode.
pattern	The pattern you want to identify. Contains events connected by event operators.
event	Events compared in the pattern.

#### Usage

The **MATCHING** clause immediately follows the **FROM** clause in a **SELECT** statement. The **FROM** clause contains the derived elements that are used as inputs for pattern matching.

**SELECT** statements containing a **MATCHING** clause cannot include any filtering or aggregation criteria.

The **MATCHING** clause consists of a mandatory interval and pattern specification.

The interval specifies the time period within which the pattern must be detected. It supports microsecond granularity and can either be represented as an interval constant (refer to the interval data type) or a parameter.

The pattern specification indicates the events or groups of events that must occur, or not occur, within the specified interval to meet the pattern matching criteria. Where a pattern specification consists of more than one event, the events or groups of events must be connected with the operators listed in the following table:

Opera- tor	Operator Name	Description
!	Not operator	Specifies a negative condition for a pattern component. Pattern con- ditions are met when the pattern component does not occur within the specified time interval. Since this is a negative condition, the pattern match is deemed successful only after the expiration of the specified time interval.
&&	Conjunction (logical AND) operator	Both pattern components linked by the conjunction operator must occur for the match condition to be met, but they do not have to occur in the order listed.
	Disjunction (Logical OR) operator	One or both pattern components linked by the Disjunction operator must occur to meet the conditions of the match. Each output row produced by a Disjunction match shows the match for one of the members of the Disjunction, and NULL values for the other mem- bers. This is true even when several members of the disjunction produce events.
,	Followed by op- erator	Pattern components linked by this operator must both occur, in the order listed, to meet the conditions of the match.

The default order of precedence in which pattern components are analyzed for a possible pattern match follows the order of operators, as they are listed in the table. The tightest binding between an operator and a pattern component is that of the Not operator. The bindings then get progressively looser, for events linked with a conjunction, disjunction, and sequence operators, respectively. This default order of precedence can be overridden by enclosing a pattern component in parentheses.

Since pattern matching on a not operator is deemed successful only after the expiration of the specified time interval, a not operator when included with a followed by operator must be its last component. This is because events succeeding the not operator will never be evaluated by the pattern rule engine owing to the expiration of the time interval.

The **MATCHING** clause of a **SELECT** statement that includes multiple derived elements in the **FROM** clause can contain an optional **ON** sub-clause, which defines one or more equality expressions that further refine the pattern matching criteria.

The equality expression is used to compare the column values of the input records or their opcodes. The left hand side of the equality can either contain a fully qualified column name, or the function <sup>1</sup>. The right hand side of the equality can contain a fully qualified column name, a constant value, or a parameter.

If the left hand side contains the function  $^2$ , the right hand side must contain a constant specifying the desired opcode. Valid opcode values are insert, update and delete.

## See also

• *Pattern Matching* on page 42

# **ON Clause: Join Syntax**

Specifies join conditions for syntax using **JOIN** terminology.

## Syntax

```
ON source1.columnA = source2.columnB [AND...]
```

## Components

source	The names of the sources in the <b>FROM</b> clause.
column	The name of the column from a particular source. Use AND when multiple column comparisons are specified. OR expressions are not supported.

## Usage

This form of the **ON** clause is required for outer and inner joins. It must consist of one or more simple comparisons, comparing a column in one data source with a column in another data source.

source1 and source2 refers to the sources (streams, windows, or delta streams) in the **FROM** clause. If aliases are used in the **FROM** clause, use the aliases rather than the actual source names.

## Restrictions

• Join conditions are limited to comparisons between columns in the two data sources of the join. The comparison cannot specify a literal value, or compare two columns in the same data source.

- Joins on page 37
- Join Examples: ANSI Syntax on page 40

<sup>&</sup>lt;sup>1</sup> getOpcode()

<sup>&</sup>lt;sup>2</sup> getOpcode()

# SELECT Clause

Specifies a projection list for a query.

## Syntax

```
SELECT { expression[AS column]}[,...]
```

## Components

expression	An expression that evaluates to a value of the same data type as the corresponding destination column.
column	the name of a column in a query destination.

## Usage

The expressions within each select list item can contain literals, column names from sources referenced in the **FROM** clause, operators, scalar functions, and parenthesis. A wild card (\*) selects all the columns from underlying sources referenced in the **FROM** clause. The AS column reference must map to a column name in the destination.

All the items in the projection must use the AS extension to map the items to the destination columns, or none of them should, in which case the assignment is performed left to right. Under some circumstances, a schema can be automatically generated for the destination, based on a query. For expressions, provide a column with the AS extension.

The **SELECT** clause inside a query specifies a select-list of one or more items. Rows from the datasources listed in the **FROM** clause are passed to the **SELECT** clause after being filtered by the **WHERE** clause, if specified. The results of the expressions in the list are processed by other clauses (if any). The query usually uses the processed select-list results as its input.

These rules apply to the select-list:

- The expression within each select-list item can contain literals, column names from one of the datasources listed in the FROM clause, operators, scalar and miscellaneous functions, and parentheses. A query select-list expression can also include aggregate functions. Alternately, you can use the "select all" (wildcard) character (\*) to specify expressions. This is equivalent to listing all column values from all datasources listed in the statement's FROM clause, from left to right, or to using data-source.\*, which is equivalent to a list of all column values from the specified data source (where data-source is the name or alias of one of the data sources listed in the FROM clause).
- These rules apply to all expressions that do not include the wildcard character:

• Each list item can specify an AS output column reference subclause indicating the column within the destination, to which the select-list item should be published. The AS subclause must be used either for all or for none of the items in the select-list.

## See also

- CREATE DELTA STREAM Statement on page 64
- DECLARE Statement on page 77
- PRIMARY KEY Clause on page 95
- SCHEMA Clause on page 97
- Chapter 5, CCL Query Construction on page 35
- FROM Clause on page 103
- AS Clause on page 88
- Scalar Functions on page 148
- Operators on page 27

# **UNION Operator**

Combines the result of two or more **SELECT** clauses into a stream or window.

## Syntax

```
{select_clause} UNION {select_clause} [ UNION ...]
```

## Components

select_clause	A <b>SELECT</b> clause.
---------------	-------------------------

## Usage

The union operation may produce a stream, delta stream, or a window.

- If the input to a union that produces a window is a stream, you must perform an aggregation operation.
- When a union joins two **SELECT** clauses, the schema of the columns selected in the two **SELECT** clauses must match.
- Ensure that a record with a particular key value is not produced by more than one input node. Otherwise, you may see duplicate rows or invalid updates.
- To be compatible, the schema for all the nodes subject to the union must have the same datatypes. However, the column names in the schemas may be different. In this case, the column names from the first **SELECT** clause are used in the schema deduction.
- If the **SELECT** statement is not a direct copy from the source, intermediate nodes are created. The compiler attempts to create delta streams or streams, but must generate windows in cases when aggregation or a **KEEP** clause.

- DECLARE blocks are not allowed for union operations.
- A node created by a union operation can have a **KEEP** clause and an **AGING** clause if the target is a window.

#### Restrictions

- The inputs to a union can be any combination of streams, delta streams, and windows.
- The inputs to a union delta stream can be a delta stream or a window, but not a stream.
- The inputs to a union window can be any combination of streams, delta streams, and windows (provided the querying involving a stream has a **GROUP BY** clause).
- A union stream or delta stream cannot have a **GROUP BY** clause specified in any of the underlying queries.

#### Examples

This example uses a union operation to produce an output stream:

```
CREATE SCHEMA MySchema (a0 integer, al STRING, a2 string);
CREATE SCHEMA MySchema2 (a0 integer, al STRING, a2 string);
CREATE INPUT STREAM InputStream1 SCHEMA MySchema;
CREATE INPUT STREAM InputStream2 SCHEMA MySchema2;
CREATE INPUT STREAM InputStream3 SCHEMA MySchema2;
CREATE OUTPUT STREAM UnionStream1 AS SELECT * FROM InputStream1
UNION
SELECT * FROM InputStream2;
```

Using a union operation to produce an output window:

```
CREATE OUTPUT WINDOW UnionWindow1
PRIMARY KEY DEDUCED
AS
SELECT in1.a0, min(in1.a1) a1, min(in1.a2) a2
FROM InputStream1 in1 GROUP BY in1.a0
UNION
SELECT in2.a0, min(in2.a1) a1, min(in2.a2) a2
FROM InputStream2 in2 GROUP BY in2.a0;
```

**Note:** Since the source is a stream and target is a window, an aggregation is specified, as is required.

This example uses a union operation to produce a delta stream:

```
CREATE DELTA STREAM Union1 PRIMARY KEY DEDUCED
AS
SELECT * FROM Stream1
UNION
SELECT a.col1, a.col2, a.col3 FROM DeltaStream1 a WHERE a.col1 >
10
UNION
SELECT a.a, sum(a.b), max(a.c) FROM Window2 GROUP BY a.a
```

## See also

- Unions on page 36
- Example: Merging Data from Streams or Windows on page 36

# WHERE Clause

Specifies a selection condition, join condition, update condition, or delete condition to filter rows of data.

## Syntax

WHERE condition | filterexpression

## Components

condition	A Boolean expression representing a selection, update, delete, or join condition, depending on the context.
filterexpression	A Boolean expression based on the columns from a stream.

## Usage

The **WHERE** clause filters rows and columns in several CCL statements, with similar syntax, but different usage and context. The **WHERE** clause:

- Specifies a selection condition for filter input from data sources in a QUERY element.
- Provides join conditions in a **FROM** clause.

## As a Selection Condition

The WHERE clause acts as a selection condition when used with a FROM clause.

The Boolean expression in this clause creates a selection that filters rows arriving in the query's data sources before passing them on to the **SELECT** clause. **WHERE** clause filtering is performed before the **GROUP BY** clause and before aggregation (if any), so it cannot include aggregate functions or the filtering of results based on the results of aggregates. You can use the **HAVING** clause for post-aggregate filtering.

The selection condition can include literals, column references from the query's data sources listed in the **FROM** clause, operators, scalar functions, parameters, and parentheses.

In a query, column references within the selection condition must refer to columns in one of the query's data sources.

## As a Join Condition

When used in conjunction with the comma-separated syntax form of the **FROM** clause, the **WHERE** clause creates one or more join condition for the comma-separated join. The use of a **WHERE** clause is optional in a comma-separated join. In the absence of a join condition, all

rows from all data sources are selected. When a **WHERE** clause is present, its syntax resembles the **ON** clause with ANSI join syntax.

The join condition can be any valid Boolean expression that specifies the condition for the join. All column references in this form of the **WHERE** clause must refer to data sources specified with the **FROM** clause.

#### As a Filter Expression

Filter expressions are supported only in input streams.

When using columns in a filter expression, use the nodeName.columnName notation. nodeName is the name of the input stream.

#### Restrictions

- A WHERE clause cannot use aggregate functions.
- A WHERE clause cannot be used with a MATCHING clause.
- Joins using the **JOIN** keyword do not use the **WHERE** clause to specify join conditions (though they can use the clause in its selection condition form).

#### Examples

This example uses a WHERE clause as a select condition:

```
CREATE INPUT WINDOW QTrades SCHEMA (
    Id integer,
    TradeTime date,
   Venue string,
    Symbol string,
    Price float,
    Shares integer
PRIMARY KEY (Id);
CREATE OUTPUT WINDOW QTradesComputeSelected
PRIMARY KEY DEDUCED
AS SELECT
    trd.*
FROM
    QTrades trd
WHERE
    trd.Symbol IN ('DELL', 'CSCO', 'SAP')
;
```

This example uses a WHERE clause as a join condition:

```
CREATE INPUT WINDOW QTrades SCHEMA (
Id integer,
TradeTime date,
Venue string,
Symbol string,
Price float,
Shares integer
)
```

```
PRIMARY KEY (Id);
CREATE OUTPUT WINDOW RecentQTrades
PRIMARY KEY DEDUCED
AS
    SELECT q.Symbol, nth(0, q.Price) Price, nth(0, q.Shares) Shares
FROM
   QTrades q
GROUP BY q.Symbol
GROUP ORDER BY q.ROWID DESC
;
CREATE INPUT WINDOW Positions
SCHEMA (BookId STRING, Symbol STRING, SharesHeld INTEGER)
PRIMARY KEY (BookId, Symbol)
;
CREATE OUTPUT WINDOW PositionValue
PRIMARY KEY (BookId, Symbol)
AS SELECT
   pos.BookId,
   pos.Symbol,
   pos.SharesHeld,
   pos.SharesHeld * q.Price Value
FROM
   Positions pos, RecentQTrades q WHERE pos.Symbol = q.Symbol
;
```

This example uses a WHERE clause as a filter expression:

```
CREATE INPUT STREAM LSETradesFiltered SCHEMA (
    Id integer,
    TradeTime date,
    Venue string,
    Symbol string,
    Price float,
    Shares integer
)
WHERE LSETradesFiltered.Symbol IN ('SAP', 'CSCO', 'DELL')
;
```

- Filtering on page 35
- *Expressions* on page 31

# CHAPTER 10 Functions

A function is a self-contained, reusable block of code that performs a specific task.

The Sybase Event Stream Processor supports:

- Built-in functions including aggregate, scalar and other functions
- User-defined SPLASH functions
- User-defined external functions

Built-in functions come with the software and include functions for common mathematical operations, aggregations, datatype conversions, and security.

#### Order of Evaluation of Operations

Operations in functions are evaluated from right to left. This is important when variables depend on another operation that must pass before a function can execute because it can cause unexpected results. For example:

integer a := 1; integer b := 2; max( a + b, ++a );

The built-in function **max()**, which returns the maximum value of a comma-separated list of values, returns 4 since ++a is evaluated first, so max(4, 2) is executed instead of max(3, 2), which may have been expected.

## See also

- Aggregate Functions on page 126
- Other Functions on page 214
- Scalar Functions on page 148
- User-Defined SPLASH Functions on page 119
- User-Defined External Functions on page 120

# **User-Defined SPLASH Functions**

Use the SPLASH programming language to write user-defined functions in either global or local declare blocks.

#### Syntax

```
DECLARE
    returnType funcName (argType argName,...) {
```

```
//function body
```

```
return value;
}
END;
```

## Usage

Function names are case-sensitive.

Functions defined at the module or project level can be used anywhere in the expressions inside that module or project. However, functions defined within streams, windows, and **FLEX** operators are visible only in the scope of those elements.

Functions are defined and there is no need to declare a function. For example, function f2 can reference f1 before f1 is defined.

# **User-Defined External Functions**

In CCL projects, use the  $\mbox{CREATE LIBRARY}$  statement to call user-defined functions written in C/C++ or Java.

Load C/C++ functions from shared libraries, .so files in Linux and UNIX, and .dll files in Windows. Load Java functions from either .class files or .jar files.

Declare external functions in CCL using the **CREATE LIBRARY** statement. Once declared, you can use the functions anywhere you use built-in functions.

**Note:** C/C++ external library calls support all datatypes, namely boolean, integer, long, float, money(n), date, bigdatetime, and binary.

Java external library calls only support integer, long, double, and string datatypes.

Complex types such as dictionaries, vectors, event caches and record types are not supported in external functions.

## **External C/C++ Function Requirements**

External C/C++ functions must conform to the interface of the Sybase Event Stream Processor by following the datatype, argument/return value, and output requirements.

## Syntax

Write the function signature to the Event Stream Processor interface:

```
int32_t funcName (int numargs,
    DataValue::DataValue * top,
    DataValue::DataValue * nextArgs,
    std::vector<void *> & arena)
```

## Datatype Requirements

The Event Stream Processor passes each function argument as a DataValue and expects to receive the return value as DataValue. The DataValue is a structure that includes all the datatypes understood by Event Stream Processor and is defined in DataValue.hpp, which is located in \$ESP\_HOME\include. The DataValue structure has this definition:

```
struct DataValue {
   union {
       bool booleanv;
       int16_t int16v;
       int32_t
                 int23v;
       int64_t int64v;
       interval_t intervalv;
       money_t moneyv;
       double doublev;
time_t datev;
       timestampval_t timestampv;
       const char * stringv;
       hirestime_t bigdatetimev;
       binary_t binaryv;
       void * objectv;
   bool null;
```

When the Boolean flag null is set to true, the value of the argument is NULL (the argument does not have a value). binary\_t is a class with two public member variables defined as:

• const uint8 t \* \_data;.

This variable points to the first byte of the data in the buffer.

• byte\_size\_t \_used;.

This variable defines the length of data used in the buffer.

**Note:** Assign memory to \_data using malloc or calloc, not new.

moneyv is a generic placeholder for money arguments with any scale; it must be told what scale a particular money argument has.

#### Argument and Return Value Requirements

Since the Event Stream Processor internal processing engine is a bytecode stack machine that keeps the top of the stack in a special location, ensure the Event Stream Processor splits function arguments into two:

• A pointer to the top of the stack of type DataValue. The top of the stack points to the last argument when more than one argument is passed to the function and to the first argument if only one argument is passed. The first argument in the interface indicates the number of arguments passed.

• A pointer to the rest of the arguments of type DataValue. The pointer points to the first argument when there is more than one argument passed to the function. It is undefined if the function has only one argument.

Note: Write the return value of the function to the top of the stack.

If the function allocates memory by calling malloc or calloc, the Event Stream Processor can release the memory after it has processed the record by adding the memory to the arena. The arena is the last argument to the function and is defined as vector of type void \*. You cannot add a pointer to the memory allocated by new to the arena; doing so can corrupt the memory and cause an unrecoverable error.

## **Output Requirement**

Ensure the function returns an error code to indicate successful completion of the function. The return value is of type int32\_t. A value of 0 indicates no error; any other values indicate an error. When an error occurs, Event Stream Processor rejects the current record.

#### See also

• *Example: Using External C/C++ Functions* on page 122

## Example: Using External C/C++ Functions

Write a C/C++ function that computes distances to the Event Stream Processor interface. After compiling the function to a shared library, declare it using the **CREATE LIBRARY** statement, and call the function as needed in your CCL project.

## Prerequisites

Know the syntax and requirements for writing C/C++ functions to the interface of the Event Stream Processor.

## Task

1. Write the function, ensuring it conforms to the Event Stream Processor interface.

For example, this function computes distance:

```
#include math.h
double distance(int numvals, double * vals){
    double sum = 0.0;
    for (int i=0; i<numvals; i++){
        sum += vals[i]*vals[i];
    }
return sqrt(sum);
}</pre>
```

To conform to the interface of the Event Stream Processor, write the function as:

#include <math.h>
#include <vector>

```
#include "DataValue.hpp"
using namespace std;
#ifdef WIN32
   #define DLLEXPORT declspec(dllexport)
#else
   #define DLLEXPORT
#endif
* This function computes the distance using the given
* arguments.
* @numargs - Number of arguments to this function.
* @top
          - Points to the last argument. Also holds the
            return value from this function.
* @nextArgs - The remaining arguments in the order provided.
 * @arena - Anything assigned to the arena is freed by the
             the server. NOTE: Do not assign return values
             to the arena. Also anything to be freed must
             be allocated using malloc only (DO NOT USE new).
extern "C" __DLLEXPORT__
int32_t distance(int numargs, DataTypes::DataValue * top,
       DataTypes::DataValue * nextArgs,
       std::vector<void *>& arena){
   double sum = 0.0;
   if (numargs <= 0){
       //Return value
       top->setDouble(0.0);
       //Return code.
      return 0;
   }
   //If any of the arguments is null result is null.
   if(top->null) return 0;
   //Top of the stack points to the last argument.
   double dist = top->val.doublev * top->val.doublev;
   //Processes the arguments from last to first.
   for(int i=numargs-2; i>=0; i--){
       //If any of the arguments is null result is null
       if((nextArgs+i)->null){
          top->null = true;
          return 0;
       }
       //accumulate the square of the distances.
       dist +=(nextArgs + i)->val.doublev * (nextArgs + i)-
>val.doublev;
  }
```

```
//Return value
top->setDouble(sqrt(dist));
//Return code.
return 0;
```

Note: Explicitly set whether the return value (top) is NULL or not.

The extern declaration ensures the function has the same name within the library and not the C++ function name.

**2.** Compile the function to a shared library.

For example, using the gcc compiler, these commands create a shared library named distance.so:

```
gcc -fPIC -shared -m64 -I.. -c -o distance.o distance.cpp gcc -fPIC -shared -m64 distance.o -o distance.so
```

3. Declare the function in the CCL project using the **CREATE LIBRARY** statement.

```
CREATE LIBRARY DistanceLib LANGUAGE C FROM 'distance.so'(
    float distance(float arg1, float arg2, float arg3);
);
```

**Note:** When searching for shared libraries (.dll files), Windows checks the path of the application. If the .dll file is not found in that directory, other directories are searched, culminating in the directories specified in the PATH environment variable.

Ensure the name of the function matches the name of the function in the library.

Call the distance function in the project using DistanceLib.distance(arg1, arg2, arg3).

## See also

• *External C/C++ Function Requirements* on page 120

## **Example: Using Java Functions**

Write a Java function that computes distances. After compiling the function as a .class or .jar file, declare it using the **CREATE LIBRARY** statement, and call the function as needed in your CCL project. Finally, link the library with the Event Stream Processor.

**Note:** The Java 1.6 runtime environment is included with Sybase Event Stream Processor. If your function requires a different version of Java, set the environment variable ESP\_JAVA\_HOME to the location of the appropriate Java virtual machine shared library. This is usually libjvm.so on Linux, UNIX, or Solaris and jvm.dll on Windows.

For example, to set the variable on a Linux, UNIX, or Solaris machine in the shell, use: export ESP\_JAVA\_HOME=/user/bin/java/jre/lib/libjvm.so 1. Write the function.

Define all functions as a public static method inside the class. For example, this function computes distances:

```
public class Distance {
    public static double distance(double arg1, double arg2,
    double arg3) {
        double sum = 0;
        sum += arg1 * arg1;
        sum += arg2 * arg2;
        sum += arg3 * arg3;
        return Math.sqrt(sum);
    }
}
```

Note: You cannot pass or return null values to external Java functions.

**2.** Compile the function to a shared library:

javac -d /home/sybase/user/java/lib Distance.java

You can also create Java archives (.jar files) of classes and refer to those when declaring the functions in the CCL project.

**3.** Declare the function and library in the CCL project using the **CREATE LIBRARY** statement.

```
CREATE LIBRARY DistanceLib LANGUAGE JAVA FROM 'Distance' (
    double distance(double arg1, double arg2, double arg3);
);
```

**Note:** 'Distance' is the name of the class. If the class is defined in a package, replace the class name with its directory, including the name.

Ensure the function signature in the library has the same name, argument datatypes, and return datatypes as the function in the .class file.

- Call the function in the project using DistanceLib.distance(arg1, arg2, arg3).
- 5. Link the Java library to the Event Stream Processor Server.

The Event Stream Processor has a built-in Java runtime environment. To link the Java function to your application, start the server with the -j option.

For .class files, specify only the directory of the file:

sp -j /home/sybase/user/java/lib

If the class is inside a .jar file located in, for example, /home/sybase/user/java, then specify the directory of the file including the file name:

```
sp -j /home/sybase/user/java/Distance.jar
```

Separate multiple paths using ":" in Linux/UNIX and ";" in Windows.

# **Aggregate Functions**

Aggregate functions operate on multiple records to calculate one value from a group of values.

The groups or rows are formed using the **GROUP BY** clause of the **SELECT** statement. The **GROUP FILTER** and **GROUP ORDER BY** clauses are used in conjunction with the **GROUP BY** clause to limit the rows in the group and to order the rows in the group respectively.

Aggregate functions, such as sum(), min() etc are allowed only in the select list and in the HAVING clause of a SELECT statement. Aggregate functions cannot be specified in the GROUP BY, GROUP ORDER BY, GROUP FILTER and WHERE clauses of the SELECT statement.

All aggregate functions ignore NULL values when performing their aggregate calculations. However, when all input passed to an aggregate function is NULL the function returns a NULL except for the count () function, which returns a 0.

Certain aggregate functions namely count(), sum(), avg() and valueInserted() are considered additive functions. Additive functions can compute its value only based upon the column values in the current event without having to look at the rest of the events in the group. A projection that uses **ONLY** additive functions allows the server to optimize the aggregation so that additional aggregation indexes are not maintained. This improves the performance of the aggregation operation considerably.

**Note:** Aggregate functions cannot be nested i.e. an aggregate function cannot be applied over an expression containing another aggregate function.

#### Example

In general, the following example shows how the aggregate functions are incorporated into CCL code:

```
CREATE INPUT WINDOW Trades
SCHEMA (TradeId LONG, Symbol, STRING, Price FLOAT, Volume LONG,
TradeTime DATE)
PRIMARY KEY (TradeId);
```

```
CREATE OUTPUT WINDOW
TradeSummary PRIMARY KEY DEDUCED
AS SELECT trd.Symbol, max(trd.Price) MaxPrice, min(trd.Price)
MinPrice, sum(trd.Volume)
TotalVolume FROM Trades trd
GROUP BY trd.Symbol;
```

- GROUP BY Clause on page 106
- GROUP FILTER Clause on page 107
- GROUP ORDER BY Clause on page 108

- HAVING Clause on page 109
- Aggregation on page 43

## any()

Aggregate. Returns a value based on an arbitrary member in a group of values.

## Syntax

```
any ( expression )
```

## Parameters

expression	An expression that will typically reference one or more columns in
	the input stream. It will be evaluated using an arbitrary member of
	the group.

## Usage

Returns the value for the expression based on an arbitrary member of the group unless the group has no members in which case a NULL value is returned. The function takes any datatype as its argument, and the function returns that same datatype.

## avg()

Aggregate. Computes the average value of a given set of arguments to identify the central tendancy of a value group.

## Syntax

avg ( numeric-expression )

## Parameters

numeric-expression	A numeric expression for which an average is computed. The ex-
	pression accepts all datatypes except boolean. The expression will
	normally reference one or more columns in a group of records such
	that the average will be computed using the reference column value
	for each member of the group.

## Usage

Compute the average value across a set of rows. The average is computed according to the following formula:



The avg function generates a 0 when a NULL value is received and takes any numeric datatype as input; returns type FLOAT.

The average function could be used to indentify things such as the average trading price of a stock over a determined period of time.

## corr()

Aggregate. Returns the correlation coefficient of a set of number pairs to determine the relationship between the two properties.

#### Syntax

corr ( dependent-expression, independent-expression )

dependent-expression	The variable that is affected by the independent variable. The expression accepts all numeric da- tatypes except timestamp, bigdatetime, and in- terval. Will normally reference one or more col- umns in the group of records to be aggregated.
independent-expression	The variable that influences the outcome. The expression accepts all numeric datatypes except timestamp, bigdatetime, and interval. Will normally reference one or more columns in the group of records to be aggregated.

## Usage

Returns the correlation coefficient of a set of number pairs. The function converts its arguments to FLOAT, performs the computation in double-precision floating point, and returns a float as the result. If the function is applied to an empty set, then it returns NULL.

Both **dependent-expression** and **independent-expression** are numeric. The function is applied to the set of (**dependent-expression**, **independent-expression**) after eliminating the pairs for which either **dependent-expression** or **independent-expression** is NULL.

$$r = \frac{\sum XY - \frac{\sum X \sum Y}{N}}{\sqrt{(\sum X^2 - \frac{(\sum X)^2}{N})(\sum Y^2 - \frac{(\sum Y)^2}{N})}}$$

where x represents the **independent-expression** and y represents the **dependent-expression**. Running totals of row\_count, sum\_x, sum\_y, sum\_xx, sum\_yy and sum\_xy are required.

The correlation function could be used to analyze the relationship between two sets of stock variables to help benchmark against competitors.

## covar\_pop()

Aggregate. Returns the population covariance of a set of number pairs to determine the relationship between the two data sets.

#### Syntax

```
covar_pop ( dependent-expression, independent-expression )
```

#### Parameters

dependent-expression	The variable that is affected by the independent variable. The expression accepts only a range of integers.
independent-expression	The variable that influences the outcome. The expression accepts only a range of integers.

## Usage

This function converts its arguments to float, performs the computation in double-precision floating point, and returns a float as the result. If the function is applied to an empty set, then it returns NULL. Both **dependent-expression** and **independent-expression** are numeric. The function is applied to the set of (**dependent-expression**, **independent-expression**) pairs after eliminating all pairs for which either **dependent-expression** or **independent-expression** is NULL. The following computation is then made:

```
(SUM(expr1 * expr2) - SUM(expr2) * SUM(expr1) / n) / n
```

where x represents the **dependent-expression**, y represents the **independent-expression**, and *n* represents the number of (x, y) pairs where neither x or y is NULL.

The covariance of a sample may be used to assess the relationship between things such as the rate of economic growth and the rate of stock market return.

## covar\_samp()

Aggregate. Returns the sample covariance of a set of number pairs.

#### Syntax

```
covar_samp ( dependent-expression, independent-expression )
```

#### Parameters

dependent-expression	The variable that is affected by the independent variable. The expression accepts only a range of integers.
independent-expression	The variable that influences the outcome. The expression accepts only a range of integers.

#### Usage

This function converts its arguments to float, performs the computation in double-precision floating point, and returns a float as the result. If the function is applied to an empty set, then it returns NULL. Both **dependent-expression** and **independent-expression** are numeric. The function is applied to the set of (**dependent-expression**, **independent-expression**) pairs after eliminating all pairs for which either **dependent-expression** or **independent-expression** is NULL.

```
(SUM(expr1 * expr2) - SUM(expr2) * SUM(expr1) / n) / (n -1)
```

Here *x* represents the **dependent-expression**, *y* represents the **independent-expression**, and *n* represents the number of (x, y) pairs where neither *x* or *y* is NULL.

The covariance of a sample may be used to indicate how two specific stocks may move together in the future, which is an important aspect before analyzing the standard deviation of a portfolio as a measure of risk.

## count()

Aggregate. Returns the number of rows in a group, excluding NULL values.

#### Syntax

count ( \* | expression )

#### Parameters

expression	A column from the source or an <b>expression</b> typically based upon
	columns from the source. It can also be a <b>constant expression</b> .

## Usage

This function counts all sets of non-NULL rows and returns a long. The function returns the number of rows in a group, excluding NULL values. Use the \* syntax to return the number of rows in the group, or use the **expression** argument to return the number of non-NULL rows.

## count(distinct)

Aggregate. Returns the number of distinct rows in a group.

## Syntax

```
count ( distinct expression )
```

## Parameters

distinct expression	A column of any datatype, except binary.
---------------------	--

## Usage

This function counts all sets of non-NULL rows and returns an integer. Duplicates are not counted. A **distinct expression** is a column or another **distinct expression** that is counted.

## exp\_weighted\_avg()

Aggregate. Calculates an exponential weighted average.

## Syntax

exp\_weighted\_avg ( expression, period-expression )

#### Parameters

expression	A numeric expression for which a weighted value is computed.
period-expression	A numeric expression specifying the period for which the average is computed.

## Usage

An exponential moving average (EMA) function applies weighting factors to values that decrease exponentially. The weighting for each older data point decreases exponentially, giving more importance to recent observations while not discarding older observations and allowing for descriptive statistical analysis.

The degree of weighting decrease is expressed as a constant smoothing factor  $\alpha$ , a number between 0 and 1.  $\alpha$  may be expressed as a percentage, so a smoothing factor of 10% is equivalent to  $\alpha$ =0.1. Alternatively,  $\alpha$  may be expressed in terms of N time periods. For example,

$$\alpha = \frac{2}{N+1}$$

N=19 is equivalent to  $\alpha=0.1$ .

The observation at a time period t is designated Yt, and the value of the EMA at any time period t is designated St. S1 is undefined. You can initialize S2 in a number of different ways, most commonly by setting S2 to Y1, though other techniques exist, such as setting S2 to an average of the first four or five observations. The prominence of the S2 initialization's effect on the resultant moving average depends on  $\alpha$ ; smaller  $\alpha$  values make the choice of S2 relatively more important than larger  $\alpha$  values, since a higher  $\alpha$  discounts older observations faster.

This type of moving average reacts faster to recent price changes than a simple moving average. The 12- and 26-day EMAs are the most popular short-term averages, and they are used to create indicators like the moving average convergence divergence (MACD) and the percentage price oscillator (PPO). In general, the 50- and 200-day EMAs are used as signals of long-term trends.

The weighted average function could be used for benchmarking over a particular time horizon.

# first()

Aggregate. Returns the first value from the group of values.

## Syntax

```
first ( expression, index )
```

expression	The function returns the same datatype as the argument.
index	(Optional) The index accepts NULL values and integer datatypes. Returns the same datatype as the argument. Which row to use, as offset from the last row in the group based on the group order by sort order. If omitted or 0, uses the last row.

## Usage

Returns the first value from a group of values. The function takes any datatype for the **expression** argument and an optional integer as the **index** argument, and returns the same datatype as the **expression**. The function performs a calculation on the specified expression and returns the first value, including NULL values.

If the argument is a pure column name, use as a scalar.

This function could be used in a first in first out (FIFO) fashion for accounts and stocks.
# first\_value()

Aggregate. Returns the first value from the group of values. Alias for first().

# last()

Aggregate. Returns the last value of a group of values.

## Syntax

last ( expression, index )

## Parameters

expression	The function returns the same datatype as the argument.
index	(Optional) The index accepts NULL values and integer datatypes. Returns the same datatype as the argument. Which row to use, as offset from the last row in the group based on the group order by sort order. If omitted or 0, uses the last row.

# Usage

Performs a calculation on the specified expression and returns the last value from a group of values. The function takes any datatype for the **expression** argument and an optional integer as the **index** argument, and returns the same datatype as the **expression**. The function performs a calculation on the specified expression and returns the first value, including NULL values.

If the argument is a pure column name, use as a scalar.

This function could be used in a last in first out (LIFO) fashion for accounts and stocks.

# last\_value()

Aggregate. Returns the last value of a group of values. Alias for last().

# lwm\_avg()

Aggregate. Returns the linearly weighted moving average for a group of values.

# Syntax

```
lwm_avg ( numeric-expression )
```

numeric-expression	Expressions include integer, long, float, money, timestamp, and	
	interval types.	

The function takes any datatype (except boolean) as its argument, and returns the same datatype. The function places more importance on the most recently received data. NULL values are not included.

An arithmetically weighted average is any average that has multiplying factors that give different weights to different data points based on time sensitivity. In technical analysis, a weighted moving average (WMA) has the specific meaning of weights which decrease arithmetically. In an *n*-day WMA, the latest day has weight *n*, the second latest n - 1, and so on, down to zero. The following equation is used to calculate the linear weighted moving average, where pM represents the price of a good on a specific time *n*.

$$WMA_M = \frac{np_M + (n-1)p_{M-1} + \dots + 2p_{M-n+2} + p_{M-n+1}}{n + (n-1) + \dots + 2 + 1}$$

Moving averages could be used to identify current trends and trend reversals based on closing numbers over a determined period of time. They also could be used to set up support and resistance levels.

# max()

Aggregate. Returns the maximum non-NULL value of a group of values.

# Syntax

max (expression)

## Parameters

expression	An expression that will typically reference one or more columns in	
	the input stream. It will be evaluated using an arbitrary member of	
	the group.	

# Usage

The returned value is based on the datatype of the input to be counted logically. If all values are NULL, the function returns NULL.

The max function can be used to assess portfolios and identify the top stocks in a group of values.

# meandeviation()

Aggregate. Returns the mean absolute deviation of a given expression over multiple rows. Absolute deviation is the mean of the absolute value of the deviations from the mean of all values.

## Syntax

```
meandeviation ( numeric-expression )
```

## Parameters

numeric-expression	An expression, commonly a column name, for which the sample-
	based standard deviation is calculated over a set of rows. The ex-
	pression will normally reference one or more columns in a group of
	records such that the mean deviation will be computed using the
	reference column value for each member of the group.

# Usage

This function converts the argument to float, performs the computation in double-precision floating point, and returns a float. The mean deviation is computed according to the following formula:

$$\sigma^2 = \frac{\Sigma(\mu - x_i)^2}{N}$$

This mean deviation does not include rows where **numeric-expression** is NULL. It returns NULL for a group containing no rows.

The mean deviation function could be used for optimization of stock portfolios on a real-time basis.

# median()

Aggregate. Returns the median value of a given expression over multiple rows to identify the central tendancy of the set of values.

## Syntax

median ( column )

## Parameter

column	Column name that accepts any datatype except binary.
--------	--

## Usage

The function returns the same datatype as the column.

Median is described as the numeric value separating the higher half of a sample, a population, or a probability distribution, from the lower half. The median of a finite list of numbers can be found by arranging all the observations from lowest value to highest value and identifying the middle value (the central tendancy). In an even number of observations, there is no single middle value; in this case the median is commonly defined as the mean of the two middle values.

The median function behaves differently for different datatypes.

- Integer the result is the average of two middle values rounded to the nearest whole number.
- Money the result is the average of two middle values.
- String the result is the first of two middle values.

The median function could be used to find the median stock price of a group of stockcodes to display the districts where variances occur between prices with the same stock.

# min()

Aggregate. Returns the minimum non-NULL value from a group of values.

#### Syntax

```
min ( expression )
```

#### Parameters

expression	An expression that will typically reference one or more columns the input stream. It will be evaluated using an arbitrary member	
	the group.	

# Usage

The returned value is based on the datatype of the input. If all values are NULL, the function returns NULL.

The min function can be used to assess portfolios and identify the lowest stocks in a group of values.

# nth()

Aggregate. Returns the nth value from a group of values. The first argument determines which value is returned.

# Syntax

nth ( number, expression )

number	An integer specifying which record in the group to reference. If no group order is specified, the default order is arrival, where 0 would be the most recent record. If group order is specified, then 0 will reference the first record in the group, 1 the next, etc
expression	An expression that references the rows in the group. This will typically include references to one or more columns in the input. Supports any datatype.

# Usage

The function returns the same datatype as its **expression** argument.

# recent()

Aggregate. Returns the most recent non-NULL value in a group of values.

## Syntax

```
recent ( expression )
```

## Parameter

expression	An expression that will typically reference one or more columns in	
	the input stream. It will be evaluated using an arbitrary member of	
	the group.	

# Usage

The function returns the same datatype used in the expression.

The recent function could be used to asses profiles on a real time basis to analyze the most current updates and changes.

# regr\_avgx()

Aggregate. Computes the average of the independent variable of the regression line.

# Syntax

```
regr_avgx ( dependent-expression , independent-expression )
```

dependent-expression	The variable that is affected by the independent
	variable. The expression accepts integer, long, float,
	timestamp, interval, and money datatypes.

independent-expression	The variable that influences the outcome. The ex-
	pression accepts integer, long, float, timestamp, in-
	terval, and money datatypes.

This function converts its arguments to float, performs the computation in double-precision floating point, and returns a float. If the function is applied to an empty set, the result is NULL. The function is applied to sets of **dependent-expression** and **independent-expression** pairs after eliminating all pairs where either variable is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, this computation is made, where *y* represents the **dependent-expression**:

avg(y)

# regr\_avgy()

Aggregate. Computes the average of the dependent variable of the regression line.

## Syntax

regr\_avgy ( dependent-expression , independent-expression )

dependent-expression	The variable that is affected by the independent variable. The expression accepts integer, long, float, timestamp, interval, and money datatypes.
independent-expression	The variable that influences the outcome. The expression accepts integer, long, float, timestamp, interval, and money datatypes.

# Usage

This function converts its arguments to float, performs the computation in double-precision floating point, and returns a float. If the function is applied to an empty set, the result is NULL. The function is applied to sets of **dependent-expression** and **independent-expression** pairs after eliminating all pairs where either variable is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, this computation is made, where *x* represents the **independent-expression**:

avg( x )

# regr\_count()

Aggregate. Returns an integer that represents the number of non-NULL number pairs used to fit the regression line.

## Syntax

```
regr_count ( dependent-expression , independent-expression )
```

#### Parameters

dependent-expression	The variable that is affected by the independent var- iable. The expression accepts integer, long, float, timestamp, interval, and money datatypes.
independent-expression	The variable that influences the outcome. The expression accepts integer, long, float, timestamp, interval, and money datatypes.

# Usage

This function counts all sets of non-NULL rows and returns a long. Rows are eliminated where one or both inputs are NULL.

# regr\_intercept()

Aggregate. Computes the y-intercept of the linear regression line that best fits the dependent and independent variables.

## Syntax

```
regr_intercept ( dependent-expression, independent-
expression )
```

## Parameters

dependent-expression	The variable that is affected by the independent variable. The expression accepts numeric datatypes, except timestamp, bigdatetime, and interval.
independent-expression	The variable that influences the outcome. The expression accepts numeric datatypes, except time-stamp, bigdatetime, and interval.

This function converts its arguments to float, performs the computation in double-precision floating point, and returns a float. If the function is applied to an empty set, the result is NULL. The function is applied to sets of **dependent-expression** and **independent-expression** pairs after eliminating all pairs where either variable is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, this

computation is made, where *x* represents the independent variable and *y* represents the dependent variable:

avg( x ) - regr\_slope( x, y ) \* avg( y )

# regr\_r2()

Aggregate. Computes the coefficient of determination (also referred to as R-squared or the goodness of fit statistic) for the regression line.

## Syntax

```
regr_r2 ( dependent-expression , independent-expression )
```

#### Parameters

dependent-expression	The variable that is affected by the independent variable. The expression accepts numeric datatypes, except timestamp, bigdatetime, and interval.
independent-expression	The variable that influences the outcome. The expression accepts numeric datatypes, except time-stamp, bigdatetime, and interval.

## Usage

This function converts its arguments to float, performs the computation in double-precision floating point, and returns a float. If the function is applied to an empty set, the result is NULL. The function is applied to sets of **dependent-expression** and **independent-expression** pairs after eliminating all pairs where either variable is NULL. The function is computed simultaneously during a single pass through the data using this formula, where *x* represents the independent variable and *y* represents the dependent variable:

```
covarPOP = ((_sum_xy * count) - (sum_x * sum_y)) * ((_sum_xy * count)
- (sum_x * sum_y))
xVarPop = (sum_xx * count) - (sum_x * sum_x)
yVarPop = (sum_yy * count) - (sum_y * sum_y)
result = covarPOP / (xvarPop * yVarPop)
```

# regr\_slope()

Aggregate. Computes the slope of the linear regression line fitted to non-NULL pairs.

## Syntax

```
regr_slope ( dependent-expression , independent-expression )
```

dependent-expression	The variable that is affected by the independent
	variable. The expression accepts numeric datatypes,
	except timestamp, bigdatetime, and interval.

independent-expression	The variable that influences the outcome. The ex-
	pression accepts numeric datatypes, except time-
	stamp, bigdatetime, and interval.

This function converts its arguments to float, performs the computation in double-precision floating point, and returns a float. If the function is applied to an empty set, the result is NULL. The function is applied to sets of **dependent-expression** and **independent-expression** pairs after eliminating all pairs where either variable is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, this computation is made, where *x* represents the independent variable and *y* represents the dependent variable:

```
covar_pop( x, y ) / var_pop( y )
```

# regr\_sxx()

Aggregate. Returns the sum of squares of independent expressions used in a linear regression model. Evaluates Use the statistical validity of a regression model.

## Syntax

```
regr_sxx ( dependent-expression , independent-expression )
```

Parameters

dependent-expression	The variable that is affected by the independent variable. The expression accepts numeric datatypes, except timestamp, bigdatetime, and interval.
independent-expression	The variable that influences the outcome. The expression accepts numeric datatypes, except time-stamp, bigdatetime, and interval.

# Usage

This function converts its arguments to float, performs the computation in double-precision floating point, and returns a float. If the function is applied to an empty set, the result is NULL. The function is applied to sets of **dependent-expression** and **independent-expression** pairs after eliminating all pairs where either variable is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, this computation is made, where *x* represents the independent variable and *y* represents the dependent variable:

regr\_count( x, y ) \* var\_pop( x )

# regr\_sxy()

Aggregate. Returns the sum of products of the dependent and independent variables. Evaluates the statistical validity of a regression model.

#### Syntax

```
regr_sxy ( dependent-expression , independent-expression )
```

dependent-expression	The variable that is affected by the independent variable. The expression accepts numeric datatypes, except timestamp, bigdatetime, and interval.
independent-expression	The variable that influences the outcome. The expression accepts numeric datatypes, except time-stamp, bigdatetime, and interval.

# Usage

This function converts its arguments to float, performs the computation in double-precision floating point, and returns a float. If the function is applied to an empty set, the result is NULL. The function is applied to sets of **dependent-expression** and **independent-expression** pairs after eliminating all pairs where either variable is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, this computation is made, where *x* represents the dependent variable and *y* represents the independent variable:

regr\_count( x, y ) \* covar\_pop( x, y )

# regr\_syy()

Aggregate. Returns values that represent the statistical validity of a regression model.

Syntax

```
regr_syy ( dependent-expression , independent-expression )
```

dependent-expression	The variable that is affected by the independent variable. The expression accepts numeric datatypes, except timestamp, bigdatetime, and interval.
independent-expression	The variable that influences the outcome. The expression accepts numeric datatypes, except time-stamp, bigdatetime, and interval.

This function converts its arguments to float, performs the computation in double-precision floating point, and returns a float. If the function is applied to an empty set, the result is NULL. The function is applied to sets of **dependent-expression** and **independent-expression** pairs after eliminating all pairs where either variable is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, this computation is made, where *x* represents the dependent variable and *y* represents the independent variable:

regr\_count( x, y ) \* var\_pop( y )

# stddev()

Aggregate. Computes the standard deviation of a sample. Alias for stddev\_samp().

# stddeviation()

Aggregate. Returns the standard deviation of a given expression over multiple rows. Alias for stddev\_samp().

# stddev\_pop()

Aggregate. Computes the standard deviation of a population consisting of a numeric-expression, as a float.

# Syntax

stddev\_pop ( numeric-expression )

## Parameters

numeric-expression	The expression, usually a column name, for which
	the population-based standard deviation is calcula-
	ted over a set of rows.

# Usage

This function converts its argument to float, performs the computation in double-precision floating point, and returns a float. The standard deviation is used to find the amount of variation between data points and the groups average. The population-based standard deviation is computed according to the following formula:

$$\sigma = \sqrt{\frac{\Sigma(\mu - x_i)^2}{N}}$$

This standard deviation does not include rows where numeric-expression is NULL. The function returns NULL for a group containing no rows.

The standard deviation of a population could be used to estimate and assess changes in securities, which could be used to establish future expectations.

# stddev\_samp()

Aggregate. Computes the standard deviation of a sample consisting of a numeric-expression, as a float.

## Syntax

```
stddev_samp ( numeric-expression )
```

## Parameters

numeric-expression	The expression, usually a column name, for which the sample-based standard deviation is calculated
	over a set of rows.

## Usage

This function converts its argument to float, performs the computation in double-precision floating point, and returns a float. The standard deviation is used to find the amount of variation between data points and the groups average. The standard deviation is computed according to the following formula, which assumes a normal distribution:

$$s = \sqrt{\frac{\Sigma(\bar{x} - x_i)^2}{n - 1}}$$

This standard deviation does not include rows where numeric-expression is NULL. The function returns NULL for a group containing either 0 or 1 rows.

The standard deviation of a sample could be used to asses the rate of return of an investment of a determined period of time.

# sum()

Aggregate. Returns the total value of the specified expression for each group of rows.

# Syntax

sum ( expression )

•	The object that is summed. The expression accepts all datatypes except boolean.
---	---

Typically, **sum** is performed on a column. The function returns the same datatype as the expression. The **sum** function uses all of the specified values and totals their values.

The sum function could be used to find the combined annual sales in order to assess long term and short term goals. By looking at the larger picture, the process of planning is simplified.

# valueinserted()

Aggregate. Returns a value including NULLS, from a group based on the last row applied into that group.

# Syntax

valueinserted ( expression )

## Parameters

expression	The expression accepts all datatypes.
------------	---------------------------------------

# Usage

This function returns the value of the expression computed using the most recent event used to insert/update the group. If the current event removes a row from the group then it returns a NULL.

This function is considered an additive function. Using only additive functions in the projection of a **SELECT** statement allows the server to optimize the aggregation, which results in greater throughput and lower memory utilization.

# var\_pop()

Aggregate. Computes the statistical variance of a population consisting of a numeric-expression, as a float.

# Syntax

var\_pop ( numeric-expression )

# Parameters

numeric-expression	A set of rows. <b>expression</b> is commonly a column
	name.

# Usage

This function converts its argument to float, performs the computation in double-precision floating point, and returns a float. The population-based variance (s2) of numeric-expression (x) is computed according to this formula:

$$s^2 = \frac{\sum (X_i - \overline{X})^2}{n}$$

This variance does not include rows where numeric-expression is NULL. The function returns NULL for a group containing no rows.

The variance of a population could be used as a measure of assessing risk.

# var\_samp()

Aggregate. Computes the statistical variance of a sample consisting of a numeric-expression, as a float.

# Syntax

var\_samp ( numeric-expression )

## Parameters

numeric-expression	A set of rows. <b>expression</b> is commonly a column name.
	nume.

# Usage

This function converts its argument to float, performs the computation in double-precision floating point, and returns a float. The variance (s2) of numeric-expression (x) is computed according to this formula, which assumes a normal distribution:

$$s^2 = \frac{\sum (X_i - \overline{X})^2}{n}$$

This variance does not include rows where numeric-expression is NULL. The function returns NULL for a group containing either 0 or 1 rows.

The variance of a sample could be used as a measure of assessing risk for a specific portfolio.

# vwap()

Aggregate. The **vwap** function computes a volume-weighted average price for a set of transactions.

# Syntax

```
vwap ( price, quantity )
```

Parameters

price	The name of the column containing the price in a set of transaction records.
quantity	The name of the column containing the number of units traded at the specified price in a set of transaction records.

**Note:** For both of these parameters, you can specify an expression containing the column name, but you must include the column name.

# Usage

The volume-weighted average price (VWAP) is a measure of the average price a stock is traded at over some period of time. For each trade, it determines the value by multiplying the price paid per share times the number of shares traded. Then it takes the sum of all these values and divides it by the sum of all the shares traded. The volume-weighted average price is computed using the following formula:

$$Pvwap = \frac{\sum j Pj \cdot Qj}{\sum j Qj}$$

The **vwap** function takes the price paid and the number of shares traded as arguments. As an input stream or window delivers trading events, the **vwap** function computes the VWAP to track the average price at which a stock has traded.

# weighted\_avg()

Aggregate. Calculates an arithmetically (or linearly) weighted average.

# Syntax

weighted\_avg ( expression )

# Parameters

expression	A numeric expression that accepts integer, long,
	float, money, timestamp, and interval datatypes.

# Usage

An arithmetically weighted average has multiplying factors that give different weights to different data points. In Event Processing, a weighted moving average (WMA) has the specific default meaning of weights which decrease arithmetically with the age of an event. So the oldest event is given the least weight and the newest event is given the most weight. The

weighted average is expressed using the following formula:

$$WMA_M = \frac{np_M + (n-1)p_{M-1} + \dots + 2p_{M-n+2} + p_{M-n+1}}{n + (n-1) + \dots + 2 + 1}$$

Where

- WMA The weighted moving averagen number of events in the group.
- **pM** Refers to the newest event.
- **pM-1** Refers to the second newest event.
- **pM-n+1** Refers to the oldest event.

The weighted average function could be used in circumstances that each value does to contribute equally to the group of values.

# xmlagg()

Aggregate. Concatenates all the XML values in the group and produces a single value.

## Syntax

xmlagg ( value )

## Parameters

value	The XML value represented as a string.
-------	--

# Usage

The function, which can be used only in aggregate streams or with event caches, returns a xmltype. Note that the xmltype cannot be stored directly in a record. To store the xml in the record you need to apply the xmlserialize function to convert the xmltype into a string.

## Example

```
xmlagg ( xmlparse (stringCol) )
```

# **Scalar Functions**

Scalar functions take a list of scalar arguments and return a single scalar value.

Different types of scalar functions include:

- Numeric functions
- String functions
- Conversion functions

- XML functions
- Date and time functions

Scalar functions take one or more expression values as arguments and return a single result value for each row of data processed by a query. These functions can appear in most expressions, and are used most often in **SELECT** clauses and **WHERE** clauses.

# See also

- SELECT Clause on page 113
- WHERE Clause on page 116

# **Numeric Functions**

Numeric functions are used with numeric values. Some numeric functions can also be used with interval and bigdatetime values. Examples of numeric functions include round () and sqrt ().

## acos()

Scalar. Returns the arccosine of a given value.

## Syntax

```
acos ( value )
```

## Parameters

value	A float between -1 and 1.
-------	---------------------------

## Usage

The function returns a float. If a value outside the range of -1 to 1 is given, the function returns NULL.

## Example

acos(0.0) returns 1.570796.

# asin()

Scalar. Returns the arcsine of a given value.

# Syntax

asin ( value )

value     A float between -1 and 1.
-------------------------------------

The function returns a float. If a value outside the range of -1 to 1 is given, the function returns NULL.

## Example

asin(1.0)returns 1.570796.

# atan()

Scalar. Returns the arctangent of a given value.

## Syntax

```
atan ( value )
```

## Parameters

value	A float.
-------	----------

# Usage

The function returns a float.

## Example

arctan(1.0) returns 0.785398.

# <u>atan2()</u>

Scalar. Returns the arctangent of the quotient of two given values.

# Syntax

```
atan2 ( value1, value2 )
```

# Parameters

value1	A float.
value2	A float.

# Usage

Returns the arctangent of the quotient of the given values, within the range of the standard arctangent function:

- If value2 > 0, then atan2 (value1, value2) returns the value of atan (value1/value2).
- If value1 >= 0 and value2 < 0, then atan2 (value1, value2) returns the value of atan (value1/value2) + pi().

- If value1 < 0 and value2 < 0, then atan2 (value1, value2) returns the value of atan (value1/value2) pi().
- If value1 > 0 and value2 = 0, then atan2 (value1, value2) returns the value of pi()/2.
- If value1 < 0 and value2 = 0, then atan2 (value1, value2) returns the value of -pi()/2.
- If value1 = value2 = 0, then atan2 (value1, value2) returns 0.

#### Example

atan2 (1, 2) returns 0.463647609, the value of atan (0.5).

## avgof()

Scalar. Returns the average value of multiple expressions, ignoring NULL parameters.

#### Syntax

```
avgof ( expression, [,...] )
```

#### Parameters

expression	There must be at least one argument, and all the arguments must be
	of the same datatype.

## Usage

If all parameters are NULL, the function returns NULL. The function accepts the following datatypes: float, integer, long, interval, money types, and date/time types.

The function returns the same datatype as its argument, however, if the expressions are numeric types (integers, floats, or longs), the function returns a float.

## Example

avgof (1, 2, NULL, 3, NULL) returns 2.0.

## bitand()

Scalar. Returns the result of performing a bitwise AND operation on two expressions.

#### Syntax

```
bitand ( expression1, expression2 )
```

expression1	Expression that simplifies to an integer or a long (must be the same
	datatype as <b>expression2</b> ).

expression2	Expression that simplifies to an integer or a long (must be the same
	datatype as <b>expression1</b> ).

The function takes the two expressions, and performs the logical AND operation on each pair of bits. The result for the pair is 1 if both bits are 1; otherwise, the result for the pair is 0. Both arguments must be the same datatype (integers or longs), and the function returns the same datatype as its arguments.

## Example

bitand (5, 3) returns 1, or in binary, bitand (101, 011) returns 001. The user cannot specify binary directly.

# bitclear()

Scalar. Returns the value of an expression after setting a specific bit to zero.

## Syntax

bitclear ( expression, bit )

## Parameters

expression	The initial value as an integer or a long.
bit	Which bit to clear, starting from 0 as the least-significant bit.

## Usage

Any **bit** argument must be an integer. The function returns the same datatype as the initial **expression** argument.

## Example

bitclear (13, 0) returns 12, or in binary, bitclear (1101, 0) returns 1100. The user cannot specify binary directly.

# bitflag()

Scalar. Returns a value with all bits set to zero, except the specified bit.

## Syntax

bitflag ( bit )

bit	An integer indicating which bit to set, starting from 0 as the least-
	significant bit.

The function returns an integer.

# Example

bitflag( 3 ) returns 8 or 1000 in binary.

# bitflaglong()

Scalar. Returns a value with all bits set to zero, except a specified bit.

## Syntax

bitflaglong ( bit )

## Parameters

bit	An integer indicating which bit to set, starting from 0 as the least-
	significant bit.

# Usage

The function returns a long.

# Example

# bitmask()

Scalar. Returns a value with all bits set to 0 except a specified range of bits.

# Syntax

```
bitmask ( first, last )
```

# Parameters

first	The first bit to set, starting from 0 as the least-significant bit.
last	The last bit to set, starting from 0 as the least-significant bit.

# Usage

Both arguments must be integers, and the function returns an integer. The order of the arguments does not matter, that is, bitmask (1, 3) yields the same result as bitmask (3, 1).

# Example

bitmask (1, 3) returns 14 or 1110 in binary.

bitmask (3, 0) returns 15 or 1111 in binary.

# bitmasklong()

Scalar. Returns a value with all bits set to 0, except a specified range of bits.

#### Syntax

bitmasklong ( first, last )

#### Parameters

first	The first bit to set, starting from 0 as the least-significant bit.
last	The last bit to set, starting from 0 as the least-significant bit.

## Usage

Both arguments must be integers, and the function returns a long.

## Example

# bitnot()

Scalar. Returns the value of an expression with all bits inverted.

## Syntax

```
bitnot ( expression )
```

## Parameters

expression	An integer or a long.
------------	-----------------------

## Usage

Returns the value of an expression after the bitwise operation is performed. Bits that were 0 become 1, and vice versa. The function returns the same datatype as the argument.

## Example

bitnot (7) returns -8, or in binary, bitnot (111) returns 111111111111111111111111111000. The user cannot specify binary directly.

# bitor()

Scalar. Returns the results of performing a bitwise OR operation on two expressions.

## Syntax

bitor ( expression1, expression2 )

expression1	Expression that simplifies to an integer or a long (must be the same as <b>expression2</b> ).
expression2	Expression that simplifies to an integer or a long (must be the same as <b>expression1</b> ).

# Usage

The function takes two bit patterns and produces another one of the same length by performing the logical OR operation on each pair. The result for the pair is 1 if the first bit or the second bit are 1, or if both bits are 1. Otherwise, the result for the pair is 0. The function returns the same datatype as its arguments.

## Example

bitor (5, 3) returns 7, or in binary, bitor (0101, 0011) returns 0111. The user cannot specify binary directly.

# bitset()

Scalar. Returns the value of an expression after setting a specific bit to 1.

## Syntax

bitset ( expression, bit )

## Parameters

expression	The initial value as an integer or a long.
bit	Which bit to set, starting from 0 as the least-significant bit.

## Usage

A **bit** argument must be an integer. The function returns the same datatype as the initial **expression** argument.

# Example

bitset (2, 3) returns 10, or in binary, bitset (0010, 3) returns 1010. The user cannot specify binary directly.

# bitshiftleft()

Scalar. Returns the value of an expression after shifting the bits left a specific number of positions.

## Syntax

bitshiftleft ( expression, count )

Parameters	
------------	--

expression	The initial value as an integer or a long. Can be an integer or a long.
count	How many positions to shift. The same number of right-most bits are set to 0. Must be an integer.

The bits that are shifted out the left are discarded, and zeros are shifted in on the right. The **expression** argument can be an integer or a long, but the **count** argument must be an integer. The function returns the same datatype as the initial **expression** argument.

## Example

bitshiftleft (10, 2) returns 40, or in binary, bitshiftleft (1010, 2) returns 101000. The user cannot specify binary directly.

# bitshiftright()

Scalar. Returns the value of an expression after shifting the bits right a specific number of positions.

# Syntax

bitshiftright ( expression, count )

## Parameters

expression	The initial value, as an integer or a long. Can be an integer or a long.
count	How many positions to shift. The same number of left-most bits are set to 0. Must be an integer.

# Usage

The bits that are shifted out the right are discarded, and zeros are shifted in on the left. The function returns the same datatype as the initial **expression** argument.

# Example

bitshiftright (3, 1) returns 1, or in binary, bitshiftright (0011, 1) returns 0001. The user cannot specify binary directly.

# bittest()

Scalar. Returns the value of a specific bit in a binary value.

## Syntax

```
bittest ( expression, bit )
```

expression	The initial value, as an integer or a long.
bit	Which bit to return. All other bits are set to zero.

## Usage

A **bit** argument must be an integer. The function returns the same datatype as the datatype of the **expression** argument.

## Example

bittest (15, 3) returns 8, or in binary, bittest (1111, 3) returns 1000. The user cannot directly specify binary.

# bittoggle()

Scalar. Returns the value of an expression after inverting the value of a specific bit.

## Syntax

```
bittoggle ( expression, bit )
```

## Parameters

expression	The initial value, as an integer or a long
bit	Which bit to toggle

# Usage

The **expression** argument can be an integer or a long, but the **bit** argument must be an integer. The function returns the same datatype as the datatype of the **expression** argument.

# Example

bittoggle (7, 3) returns 15, or in binary, bittoggle (0111, 3) returns 1111. The user cannot specify binary directly.

# bitxor()

Scalar. Returns the results of performing a bitwise exclusive OR (XOR) operation on two expressions.

# Syntax

bitxor ( expression1, expression2 )

expression1	Expression that simplifies to an integer or a long (must be the same datatype as <b>expression2</b> )
expression2	Expression that simplifies to an integer or a long (must be the same datatype as <b>expression1</b> )

# Usage

The function performs the logical XOR operation on each pair of corresponding bits. The result for the pair of bits is 1 if the two bits are different, or 0 if they are the same. Using **bitxor()** on the same expression yields 0. The function returns the same datatype as its arguments.

# Example

bitxor (3, 3) returns 0.

bitxor (10, 15) returns 5, or in binary, bitxor (1010, 1111) returns 0101. The user cannot specify binary directly.

# cbrt()

Scalar. Returns the cube root of a number.

## Syntax

cbrt ( value )

## Parameters

value	A numeric datatype
-------	--------------------

# Usage

The function returns a float. If the argument is invalid, the server logs a Floating-point exception error.

# Example

cbrt (1000.00) returns 10.0.

# ceil()

Scalar. Rounds a number up to the nearest whole number..

# Syntax

ceil ( value )

value	A float or money type
-------	-----------------------

## Usage

The function returns the same datatype as the argument.

#### Example

ceil (100.20) returns 101.0.

## compare()

Scalar. Determines which of two values is larger.

#### Syntax

compare ( value1, value2 )

#### Parameters

value1	Any datatype
value2	Any datatype

#### Usage

The function returns an integer (1, -1, or 0). If the first value is larger, the function returns 1. If the second value is larger, the function returns -1. If they are equal, it returns 0.

## Example

compare ( (asin(0.5), (acos(0.5) ) returns -1.

## cos()

Scalar. Returns the cosine of a given value expressed in radians.

#### Syntax

```
cos ( value )
```

#### Parameters

value	A float
-------	---------

Usage The function returns a float.

*Example* cos (0.5) returns 0.87758.

## cosd()

Scalar. Returns the cosine of a given value, expressed in degrees.

#### Syntax

```
cosd ( value )
```

## Parameters

value	A float
-------	---------

## Usage

The function returns a float.

## Example

cosd (90.0) returns -0.448073616.

# cosh()

Scalar. Returns the hyperbolic cosine of a given value expressed in radians.

#### Syntax

```
cosh ( value )
```

## Parameters

value	A float
-------	---------

## Usage

The function returns a float.

## Example

cosh (0.5) returns 1.12762597.

# distance()

Scalar. Returns a value representing the distance between two points in two or three dimensions.

## Syntax

```
distance ( pointlx, pointly, [pointlz], point2x, point2y,
[point2z] )
```

point1x	An expression that evaluates to a value representing the position of
	the first point on the x axis.

point1y	An expression that evaluates to a value representing the position of the first point on the y axis.
point1z	An expression that evaluates to a value representing the position of the first point on the z axis.
point2x	An expression that evaluates to a value representing the position of the second point on the x axis.
point2y	An expression that evaluates to a value representing the position of the second point on the y axis.
point2z	An expression that evaluates to a value representing the position of the second point on the z axis.

Returns a number representing the distance between two points in either two or three dimensions. All arguments must be the same numeric type, and the function returns the same datatype.

## Example

distance (7.5, 6.5, 10.5, 10.5) returns 5.0. distance (1.2, 3.4, 5.6, 7.8, 9.10, 11.12) returns 10.320872.

# distancesquared()

Scalar. Returns a number representing the square of the distance between two points in either two or three dimensions.

## Syntax

```
distancesquared ( pointlx, pointly, [pointlz], point2x, point2y, [point2z] )
```

point1x	An expression that evaluates to a value representing the position of the first point on the x axis.
point1y	An expression that evaluates to a value representing the position of the first point on the y axis.
point1z	An expression that evaluates to a value representing the position of the first point on the z axis.
point2x	An expression that evaluates to a value representing the position of the second point on the x axis.

point2y	An expression that evaluates to a value representing the position of the second point on the y axis.
point2z	An expression that evaluates to a value representing the position of the second point on the z axis.

Returns a number representing the square of the distance between two points in either two or three dimensions. All arguments must be of the same numeric type, and the function returns the same datatype.

# Example

```
distancesquared (7.5, 6.5, 10.5, 10.5) returns 25.0.
```

```
distancesquared (1.2, 3.4, 5.6, 7.8, 9.10, 11.12) returns 106.502400.
```

# floor()

Scalar. Rounds a number down.

## Syntax

```
floor ( value )
```

## Parameters

value	A float or a money type.
-------	--------------------------

## Usage

Rounds a given number down to the nearest whole number. The function takes a float or a money type, and the function returns the same datatype as its argument.

## Example

floor (100.20) returns 100.0.

floor ( 1.56) returns 1.0.

# isnull()

Scalar. Determines if an expression is NULL.

## Syntax

```
isnull ( expression )
```

expression	An expression of any datatype.
------------	--------------------------------

Determines if an expression is NULL. The function can take any datatype as its argument, and the function returns an integer. The function returns 1 if the argument is NULL, and 0 otherwise.

# Example

```
is null ('examplestring') returns 0.
```

# length()

Scalar. Returns the number of bytes of a given binary value.

# Syntax

```
length ( binary )
```

# Parameters

binary	A binary value.
--------	-----------------

# Usage

Returns the number of bytes that make up a given binary value. The function takes a binary value as its argument, and the function returns an integer. If the binary value is NULL, the function returns NULL.

# Example

```
length (hex_binary ('0xaa1234')) returns 3.
```

```
length (hex_binary ('aa')) returns 1.
```

# <u>In()</u>

Scalar. Returns the natural logarithm of a given number.

# Syntax

```
ln ( value )
```

# Parameters

value     A float.
--------------------

# Usage

Returns the natural logarithm of a number. If the argument is invalid (for example, less than 0), the server logs a "Floating-point exception" error. The function takes a float as its argument, and the function returns a float.

## Example

ln (2.718281828) returns 1.0.

# log2()

Scalar. Returns the logarithm of a given value to the base 2.

## Syntax

log2 ( value )

## Parameters

value
-------

# Usage

Returns the logarithm of a given value to the base 2. The function expects a float for its argument, however, an integer will be promoted to a float when the function executes. The function returns a float.

## Example

log2 (8.0) returns 3.0.

# log10()

Scalar. Returns the logarithm of a given value to a base of 10.

# Syntax

log10 ( value )

## Parameters

value An	expression that evaluates to a float greater than or equal to 0.
----------	--

# Usage

Returns the logarithm of a given value to a base of 10. The function expects a float as it argument, however, an integer will be promoted to a float when the function executes. The function returns a float.

## Example

log (100.0) returns 2.0.

# logx()

Scalar. Returns the logarithm of a given value to a specified base.

# Syntax

logx ( value, base )

## Parameters

value	An expression that evaluates to a float greater than or equal to 0.
base	An expression that evaluates to a float greater than 1.

# Usage

Returns the logarithm of a given value to a specified base. The function expects floats for its arguments, however, integers will be promoted to floats when the function executes. The function returns a float.

# Example

logx (8.0, 2.0) returns 3.0.

# maxof()

Scalar. Returns the maximum value from a list of expressions.

# Syntax

maxof ( expression [,...] )

## Parameters

expression	There must be at least one argument, and all the arguments must be
	of the same datatype.

# Usage

Returns the maximum value from a list of expressions. NULL values are ignored. If all of the arguments are NULL, the function returns NULL. The arguments can be of any datatype, but they must be of the same datatype. The function returns the same datatype as its arguments.

# Example

maxof (1.34, 3.35, 10.93, NULL) returns 10.93.

# minof()

Scalar. Returns the minimum value from a list of expressions.

## Syntax

```
minof ( expression [,...] )
```

expression	There must be at least one argument, and all the arguments must be
	of the same datatype.

## Usage

Returns the minimum value from a list of expressions. NULL values are ignored. If all of the arguments are NULL, the function returns NULL. The arguments can be of any datatype, but they must be of the same datatype. The function returns the same datatype as its arguments.

## Example

min ( 0.61, NULL, 2.34, 1.32) returns 0.61.

# nextval()

Scalar. Returns a value larger than that returned by the previous call. The first call returns 1.

## Syntax

nextval()

# Usage

The first call to the function returns 1, and then each subsequent call returns a value larger than that returned by the previous call. The increase in the values is not necessarily one; it may be larger. Each call to **nextval()** returns a new value, even if it is called more than once in a single statement. The function takes no arguments, and the function returns a long.

# Example

The first call to nextval() returns 1. Calling nextval() a second time could return 14, for example.

# <u>pi()</u>

Scalar. Returns a numerical approximation of the constant pi.

# Syntax

pi()

# Usage

Returns a numerical approximation of the constant pi. The function does not take any arguments, and the function returns a float.

# Example

pi() returns 3.141593.

# power()

Scalar. Returns the value of a given base raised to a specified exponent.

## Syntax

power ( base, exponent )

#### Parameters

base	Any numeric type.
exponent	Float that specifies the number that the <b>base</b> will be raised to.

## Usage

Returns the value of a given base raised to a specified exponent. The function takes a numeric type for the **base** argument, but the **exponent** must be a float. The function returns the same datatype as the **base** argument.

#### Example

power (2.0, 3.0) returns 8.0.

# random()

Scalar. Returns a random value greater than or equal to 0 and less than 1.

## Syntax

random()

## Usage

Returns a random value greater than or equal to 0 and less than 1. The function does not take any arguments, and the function returns a float.

## Example

random() may return 0.54 on a call, for example.

## round()

Scalar. Returns a number rounded to the specified number of digits.

## Syntax

round ( value, digits )

value	A float representing a value that needs to be rounded.
digits	The number of digits after the decimal point to round the value to.

Returns a number rounded to the specified number of digits. The value is rounded to the number of decimal points specified by the **digits** argument. The function follows standard rounding rules. Both arguments must be floats, and the function returns a float.

## Example

round (66.778, 1) returns 66.8.

# sign()

Scalar. Determines whether a given value is positive or negative.

## Syntax

```
sign ( value )
```

## Parameters

value	Any type that can have a sign (integer, float, long, interval, money).
-------	--

# Usage

Determines whether a given value is positive or negative. The function returns 1 if the value is positive, -1 if the value is negative, and 0 otherwise. The argument can be any type that has a sign, and the function returns an integer.

# Example

```
sign ( cosd(45.0) ) returns 1.
```

# sin()

Scalar. Returns the sine of a given value.

## Syntax

```
sin ( value )
```

## Parameters

value	A float.
-------	----------

# Usage

Returns the sine of a given value, expressed in radians. The function takes a float as its argument, and the function returns a float.

# Example

```
sin ( pi() ) returns 0.
```
## <u>sinh()</u>

Scalar. Returns the hyperbolic sine of a given value.

### Syntax

sinh ( value )

### Parameters

value	A float.
-------	----------

### Usage

Returns the hyperbolic sine of a given value, expressed in radians. The function takes a float as its argument, and the function returns a float.

### Example

sinh (0.5) returns 0.521095305.

## sqrt()

Scalar. Returns the square root of a given number.

### Syntax

```
sqrt ( value )
```

#### Parameters

value	A money or numeric type.
-------	--------------------------

### Usage

Returns the square root of a given number. The function takes a numeric type or a money type as its argument, and the function returns a float. If the argument is invalid, the function returns a "Floating-point exception" error.

### Example

sqrt (100.0) returns 10.0.

### tan()

Scalar. Returns the tangent of a given value.

### Syntax

tan ( value )

#### Parameters

value	A float.
-------	----------

### Usage

Returns the tangent of a given value, expressed in radians. The function takes a float as its argument, and the function returns a float.

#### Example

tan (0.0) returns 0.

### tand()

Scalar. Returns the tangent of a given value, expressed in degrees.

#### Syntax

tand ( value )

#### Parameters

value	A float.
-------	----------

### Usage

Returns the tangent of a given value, expressed in degrees. The function takes a float as its argument, and the function returns a float.

### Example

tand (45.0) returns 1.61977519.

### tanh()

Scalar. Returns the hyperbolic tangent of a given value.

#### Syntax

tanh ( value )

#### Parameters

value A float.	
----------------	--

### Usage

Returns the hyperbolic tangent of a given value. The function takes a float as its argument, and the function returns a float.

#### Example

tanh (0.5) returns 0.462117157.

# **String Functions**

String functions are used with STRING values and usually return a STRING value. Examples of string functions include left (), rtrim (), and replace ().

## int32()

Scalar. Converts a given string into an integer.

### Syntax

int32 ( string )

### Parameters

string	A string that starts with an optional minus sign and contains only digits.
	-

### Usage

Converts a given string into an integer. The function takes a string as its argument, and the function returns an integer. An invalid string causes the function to return NULL.

### Example

int32 ('1935') returns 1935.

## left()

Scalar. Returns a specified number of characters from the beginning of a given string.

### Syntax

left ( string, count )

### Parameters

string	A string.
count	The number of characters to return.

## Usage

Returns a specified number of characters from the beginning of a given string. The function takes a string and an integer as the **count** argument. The function returns a string. If **count** is a negative number, the function returns NULL. If **count** is 0, the function returns an empty string.

The function works with UTF-8 strings if the -U server option is specified.

### Example

left ('examplestring', 7) returns 'example'.

### like()

Scalar. Determines whether a given string matches a specified pattern string.

#### Syntax

like ( string, pattern )

#### Parameters

string	A string.
pattern	A pattern of characters, as a string. Can contain wildcards.

#### Usage

Determines whether a string matches a pattern string. The function returns 1 if the string matches the pattern, and 0 otherwise. The pattern argument can contain wildcards: '\_' matches a single arbitrary character, and '%' matches 0 or more arbitrary characters. The function takes in two strings as its arguments, and returns an integer.

Note: In SQL, the infix notation can also be used: sourceString like patternString.

#### Example

like ('MSFT', 'M%T') returns 1.

### lower()

Scalar. Returns a new string where all the characters of the given string are lowercase.

#### Syntax

```
lower ( string )
```

#### Parameters

string	A string.
--------	-----------

#### Usage

Returns a string where all the characters of a given string are lowercase. The function takes a string as its argument, and the function returns a string.

#### Example

upper ('This Is A Test') returns 'this is a test'.

## ltrim()

Scalar. Trims spaces from the left side of a string.

### Syntax

```
ltrim ( string )
```

### Parameters

string	A string.
--------	-----------

### Usage

Trims spaces from the left side of the string. The function takes a string as its argument, and the function returns a string.

### Example

```
ltrim (' examplestring') returns 'examplestring'.
```

### patindex()

Scalar. Determines the position of the nth occurrence of a pattern within a source string.

### Syntax

```
patindex ( string, pattern, number [, position] [,
constant_string] )
```

#### Parameters

string	A source string.
pattern	String representing the pattern to search for.
number	Occurence of the pattern to look for.
position	(optional) Starting position (0 based index) of the search. Default is 0.
constant_string	(optional) Boolean indicating whether the pattern argument should be treated as a constant string instead of a pattern. Default is false.

### Usage

Determines the position of the nth occurrence of a pattern within a source string. The pattern can contain wildcards: "\_" matches a single arbitrary character; "%" matches 0 or more arbitrary characters. If fewer than n instances of the pattern are found in the string, the function returns -1.

The function takes strings for the **string** and the **pattern** arguments, and integers for the **number** and **position** arguments. The **constant\_string** argument is a Boolean. The function

returns an integer representing the position of the nth occurrence of the pattern within the given string.

If number is less than or equal to zero, the function returns NULL. If **position** is less than 0, the function starts searching from the start of the string. If position is greater than the length of the **string** argument, **patindex()** returns -1.

The function works with UTF-8 strings if the -U server option is specified.

```
Example
patindex('longlongstring', 'long', 2) returns 4.
patindex('longstring', 'long', 2) returns - 1.
patindex('String', __n, 1) returns 2.
patindex('String', %n, 1) returns 0.
patindex('String', __n, 1, false) returns 2.
patindex('String', __n, 1, true) returns -1.
patindex('String', S, 1, 0, false) returns 0.
patindex('String', i, 2, 2, true) returns 6.
```

### real()

Scalar. Converts a given string into a float.

#### Syntax

real ( string )

#### Parameters

string	A valid string must be a sequence of digits, optionally containing a
	decimal-point character. The input may also include an optional
	minus sign as the first character, or an optional exponent part,
	which itself consists of an 'e' or 'E' character followed by an op-
	tional sign and a sequence of digits.

#### Usage

Converts a given string into a float. The function takes a string as its argument, and the function returns a float. An invalid string causes the function to return NULL.

#### Example

real ('43.4745') returns 43.4745.

### regexp\_firstsearch()

Scalar. Returns the first occurrence of a POSIX regular expression pattern found in a given string.

#### Syntax

```
regexp_firstsearch ( string, regex )
```

#### Parameters

string	A string.
regex	A POSIX regular expression pattern. This pattern is limited to the Perl syntax.

#### Usage

Returns the first occurrence of a POSIX regular expression pattern found in a given string. If string does not contain a match for the pattern, or if the specified pattern is not a valid regular expression, the function returns NULL. One or more subexpressions can be included in the pattern, each enclosed in parentheses. If string contains a match for the pattern, the function only returns the parts of the pattern specified by the first subexpression. The function returns a string.

The function works with UTF-8 strings if the -U server option is specified.

#### Example

```
regexp_firstsearch('aaadogaaa', '[b-z]*') returns 'dog'.
regexp_firstsearch('h', '[i-z]*') returns NULL.
regexp_firstsearch('aaaaabaaaabbbaaaa', '[b-z]*') returns 'b'.
```

#### regexp\_replace()

Scalar. Returns a given string with the first occurrence of a match for a POSIX regular expression pattern replaced with a second, specified string.

#### Syntax

```
regexp_replace ( string, regex, replacement )
```

string	A string.
regex	A POSIX regular expression pattern. This pattern is limited to the Perl syntax.
replacement	A string to replace the part of the string that matches regex.

Returns a given string with the first occurrence of a match for a POSIX regular expression pattern replaced with a second, specified string. If string does not contain a match for the POSIX regular expression, the function returns the string with no replacements. If **regex** is not a valid regular expression, the function returns NULL.

The function works with UTF-8 strings if the -U server option is specified.

#### Example

```
regexp_replace('aaadogaaa', '[b-z]*', 'cat') returns 'aaacataaa'.
regexp_replace('aaadogaaa', '[b-z]*', '') returns 'aaaaaa'.
regexp_replace('aaa', '[a-z]*', 'dog') returns 'dog'.
regexp_replace('aaa', '[b-z]*', 'dog') returns 'aaa'.
```

### regexp\_search()

Scalar. Determines whether or not a string contains a match for a POSIX regular expression pattern.

### Syntax

```
regexp_search ( string, regex )
```

#### Parameters

string	A string.
regex	A POSIX regular expression pattern. This pattern is limited to the Perl Syntax.

### Usage

Determines whether or not a string contains a match for a POSIX regular expression pattern. The function returns the Boolean expression corresponding to whether or not the string contains the pattern (TRUE or FALSE).

The function works with UTF-8 strings if the -U server option is specified.

### Example

```
regexp_search('aaadogaaa', '[b-z]*') returns TRUE.
regexp_search('h', '[i-z]*') returns FALSE.
```

### replace()

Scalar. Returns a new string where all the occurrences of the second string in the first string are replaced with the third string.

#### Syntax

```
replace ( target, substring, repstring )
```

#### Parameters

target	A string.
substring	The string of characters to replace.
repstring	The replacement for the characters, as a string.

#### Usage

Returns a new string where all the occurrences of the second string in the first string are replaced with the third string. The function takes three string arguments, and returns a string.

#### Example

```
replace ('NewAmsterdam', 'New', 'Old') returns 'OldAmsterdam'.
```

### right()

Scalar. Returns the rightmost characters of a string.

#### Syntax

right ( string, number )

#### Parameters

string	A string.
number	The number of characters to return from the string.

#### Usage

Returns the rightmost characters of a string. The function takes in a string and an integer, and returns a string.

#### Example

right ('examplestring', 6) returns 'string'.

### rtrim()

Scalar. Trims spaces from the right of a string.

### Syntax

rtrim ( string )

### Parameters

string	A string.
--------	-----------

### Usage

Trims the spaces from the right side of the string. The function takes in a string as its argument, and returns a string.

### Example

```
rtrim ('examplestring ') returns 'examplestring'.
```

### string()

Scalar. Converts a given value of any type to an equivalent string.

### Syntax

```
string ( value )
```

#### Parameters

value	An argument of any datatype, except binary or string.
-------	---

#### Usage

Converts a given value into an equivalent string expression. The argument can be any datatype, except binary or string. The function returns a string.

### Example

string (1935) returns '1935'.

### substr()

Scalar. Returns a substring of a given string, based on a start position and number of characters.

#### Syntax

```
substr ( string, position, number )
```

Parameters

string	A string.
position	The starting position to start taking a substring. The first character or space in a string is in position 0.
number	The number of characters in the substring.

Returns a substring of a given string, based on a start position and number of characters. The first argument must be a string, and the **position** and **number** arguments must be integers. The function returns a string.

### Example

```
substr ('thissubstring', 4, 3) returns 'sub'.
```

## trim()

Scalar. Returns a given string after removing trailing and leading spaces.

### Syntax

```
trim ( string )
```

#### Parameters

string	A string. Works with UTF-8 strings.
--------	-------------------------------------

### Usage

Returns a given string after removing trailing and leading spaces. The function takes a string as the argument, and returns a string. The function returns the same value as applying **ltrim()** and **rtrim()** to a given string.

## Example

```
trim (' examplestring ') returns 'examplestring'.
trim(' ') returns ".
trim('a') returns 'a'.
```

## trunc()

Scalar. Truncates the time portion of a date to 00:00:00 and returns the new date value.

### Syntax

```
trunc ( datevalue )
```

#### Parameters

datevalue	A date or bigdatetime.
-----------	------------------------

#### Usage

Truncates the time portion of a date value to 00:00:00 and returns the new date value. The function takes a date or bigdatetime as its argument, and the function returns the same datatype.

#### Example

```
trunc (undate ('2001:05:23 12:34:64')) returns 2001:05:23 00:00:00.
```

### upper()

Scalar. Returns a string where all the characters of a given string are uppercase.

#### Syntax

upper ( string )

#### Parameters

string	A string.
--------	-----------

#### Usage

Returns a string where all the characters of a given string are uppercase. The argument of the function is a string, and the function returns a string.

#### Example

upper ('This Is A Test') returns 'THIS IS A TEST'.

## **Conversion Functions**

Conversion functions convert data values of various datatypes to the datatype specified by the function name.

#### ascii()

Scalar. Returns the Unicode code point for a particular character, or the UTF-8 code point if the -U server option is specified.

#### Syntax

ascii ( character )

character	A character string.
-----------	---------------------

If empty or NULL, the function returns NULL. Otherwise, the function returns the code point as an integer.

### Example

ascii ('D') returns 68.

ascii ('Dog') also returns 68 since only the first character is converted.

### base64\_binary()

Scalar. Returns a binary value for a given base64-encoded string.

### Syntax

base64\_binary ( string )

### Parameters

string	A base 64-encoded string. Valid characters include a-z, A-Z, 0-9, /, and +.

### Usage

The function converts a base64-encoded string to a binary type. The string length cannot have a remainder of 1 when divided by 4, as it makes the encoding invalid. Optionally, use one or two padding characters, '=' in order to make the length divisible by 4.

### Example

```
base64_binary ( 'bGVhc3VyZS4=') returns 6C6561737572652E.
base64 binary ( 'ZQ==') returns 65.
```

## base64\_string()

Scalar. Returns a base64-encoded string for a given binary value.

#### Syntax

base64\_string ( binary )

#### Parameters

binary A binary value.	
------------------------	--

### Usage

The function encodes a binary value to form a base64-encoded string. One or two padding characters, '=' are added to the end to make the string length divisible by 4. The function returns a string.

### Example

```
base64_string ( hex_binary ('64') ) returns ZQ==. 
 base64_string ( hex_binary ('6C6561737572652E') ) returns bGVhc3VyZS4=.
```

### cast()

Scalar. Converts the value of one datatype to another datatype allowing overflows and truncation.

### Syntax

```
cast ( type, number )
```

### Parameters

type	Any datatype, except binary or string.
number	A datatype that can be cast to the new specified datatype.

## Usage

The type argument must be a numeric type, money type, or a date/time type. You can cast expressions of any type except binary or string types.

Casting from larger types to smaller types may cause overflow. Casting from decimal types (like float or money) to nondecimal types (like integer) truncates the decimal portion. Both overflows and truncation are allowed. Use this function to force a cast in places where an implicit cast is disallowed, such as when converting an integer to a long.

When comparing values of varying scale, cast one value to the other to make the two values compatible. For example, you can compare money values of different scale only by casting to a common type.

How to cast money values of different scale depends on how you compare the two values:

- If you set 100.55D2, a money(2) type, as greater than (>) 100.545D3, a money(3) type, the result is false because the values are represented internally without the decimal point. Therefore, 10055 cannot be greater than 100545. In this example, you can perform casting on either value to produce a true result. When you cast 10055 to 100545, the comparison becomes 100550>100545, which is true. When you cast 100545 to 10055, the comparison becomes 10055>10054, which is also true.
- If you set 100.55D2 as equal (=) to 100.556D3, the result is false. In this example, the result changes depending on which value you cast. When you cast 10055 to 100556, the comparison becomes 100550=100556, which is false. When you cast 100556 to 10055, the comparison becomes 10055=10055, which is true.

You may prefer to cast lower scale values to higher scale values to avoid incorrect comparison results and to maintain scale.

### Example

```
cast ( integer, 1.23) returns 1.
```

## <u>char()</u>

Scalar. Returns the characters responding to one or more Unicode code points, or the UTF-8 code points if the -U server option is specified.

### Syntax

```
char ( expression [,...] )
```

### Parameters

expression	One or more Unicode code points. The arguments must be integers.
------------	--

#### Usage

An invalid code point, 0, or NULL returns NULL. The function returns a string.

### Example

char (68) returns 'D'.

char(68, 68, 68) returns 'DDD'.

### concat()

Scalar. Returns the binary value of the concatenation of two given binary values.

#### Syntax

```
concat ( binary1, binary2 )
```

#### Parameters

binary1	A binary value
binary2	A binary value

#### Usage

The function returns a binary value. The function returns NULL if either argument is NULL.

#### Example

```
concat ( hex_binary ('aabbcc'), hex_binary ('ddeeff') ) returns
AABBCCDDEEFF.
```

```
concat ( hex_binary ('ddeeff'), hex_binary ('aabbcc') )returns
DDEEFFAABBCC.
```

### extract()

Scalar. Extracts and returns a portion of a given binary value.

#### Syntax

```
extract ( binary, startByte, numberOfBytes )
```

#### Parameters

binary	A binary value.
startByte	Integer representing the starting position for the extraction.
numberOfBytes	Integer representing the length of the extraction.

### Usage

Extracts a binary value starting at the **startByte** argument for a specified length. The function takes a binary value and two integers as its arguments (for **startByte** and **numberOfBytes**), and the function returns a binary value.

For example, if a binary value was composed of bytes abcde, extract ( bytes, 2,3 ) would produce cde. If length goes past end of binary value the rest of the binary value is returned. In the previous example, extract ( bytes, 2, 4 ) would still return cde.

### Example

```
extract ( hex_binary ('alb2c3e4'),1,2)returns B2C3.
extract ( hex_binary ('alb2c3e4'),3,1) returns E4.
extract ( hex_binary ('alb2c3e4'),0,4)returns AlB2C3E4.
```

### fromnetbinary()

Scalar. Converts a binary in network byte order to an integer in host byte order.

### Syntax

```
fromnetbinary ( binary )
```

#### Parameters

binary	A binary in network byte order.
--------	---------------------------------

#### Usage

Takes a binary in network byte order and converts it to an integer in host byte order. Works for positive and negative values. The function takes a binary value as its argument and the function returns an integer. The function returns an error if the binary value is more than 4 bytes long.

**Example** fromnetbinary (FFFFFF6) returns -10.

fromnetbinary (0012ADE4) returns 1224164.

### hex\_binary()

Scalar. Converts a hex string into a binary type.

#### Syntax

```
hex_binary ( string )
```

#### Parameters

string	A hex string, with or without the preceding "0x" or "0X".
--------	---

### Usage

Takes a hex string, and converts it into a binary type. Valid characters for a hex string are a-f, A-F, and 0-9. The string must contain an even number of characters. The function takes a string as its argument, and the function returns a binary value.

### Example

hex\_binary ('0xAA1B223F') returns AA1B223F.

hex\_binary ('0xaa') returns AA.

### hex\_string()

Scalar. Converts a binary value into a hex string.

#### Syntax

```
hex_string ( binary )
```

#### Parameters

binary	A binary value.
--------	-----------------

### Usage

Converts a binary value into a hex string. The function takes a binary value as its argument, and the function returns a string that represents a hex string without the preceding "0x" in all uppercase.

### Example

```
hex_string ( hex_binary ('0xaa') ) returns AA.
hex_string ( hex_binary ('0xaa1234') ) returns AA1234.
```

### msecToTime()

Scalar. Converts a given number of milliseconds to a bigdatetime.

### Syntax

msecToTime ( milliseconds )

### Parameters

milliseconds	A long representing the number of milliseconds since the epoch
	(midnight, January 1, 1970 UTC).

### Usage

Converts a given number of milliseconds to a bigdatetime. The function takes a long as its argument, and the function returns a bigdatetime.

### Example

```
msecToTime (3661001) returns 1970-01-01 01:01:01.001.
```

## secToTime()

Scalar. Converts a given number of seconds to a bigdatetime.

#### Syntax

secToTime ( seconds )

#### Parameters

seconds	A long representing the number of seconds since the epoch (mid-
	night, January 1, 1970 UTC).

#### Usage

Converts a given number of seconds to a bigdatetime. The function takes a long as its argument, and the function returns a bigdatetime.

### Example

secToTime (3661) returns 1970-01-01 01:01:01.000000.

### timeToMsec()

Scalar. Converts a bigdatetime to the number of milliseconds since the epoch (midnight, January 1, 1970).

### Syntax

timeToMsec ( time )

#### Parameters

time	A bigdatetime.
------	----------------

#### Usage

Converts a bigdatetime to the number of milliseconds since the epoch (midnight, January 1, 1970). The function takes a bigdatetime as its argument, and the function returns a long representing the number of milliseconds since the epoch (midnight, January 1, 1970 UTC). The function truncates the microseconds that are part of the bigdatetime.

#### Example

```
timeToMsec ( unbigdatetime('1970-01-01 01:01:01:002100') )
returns 3661002.
```

### timeToUsec()

Scalar. Converts a bigdatetime to the number of microseconds since the epoch (midnight, January 1, 1970).

### Syntax

timeToUsec ( time )

#### Parameters

time	A bigdatetime.
------	----------------

#### Usage

Converts a bigdatetime to the number of microseconds since the epoch (midnight, January 1, 1970). The function takes a bigdatetime as its argument, and the function returns a long representing the number of microseconds since the epoch (midnight, January 1, 1970 UTC).

#### Example

```
timeToUsec ( unbigdatetime ('1970-01-01 01:01:01.000001')) returns 3661000001.
```

### timeToSec()

Scalar. Converts a bigdate time to the number of seconds since the epoch (midnight, January 1, 1970).

### Syntax

timeToSec ( time )

time	A bigdatetime.
------	----------------

Converts a bigdatetime to the number of seconds since the epoch (midnight, January 1, 1970). The function takes a bigdatetime as its argument, and the function returns a long representing the number of seconds since the epoch (midnight, January 1, 1970 UTC). The function truncates the milliseconds or microseconds that are part of the bigdatetime.

### Example

```
timeToSec ( unbigdatetime('1970-01-01 01:01:01:000000') )
returns 3661.
```

### to\_binary()

Scalar. Converts a given value to a binary value.

### Syntax

to\_binary ( value )

#### Parameter

value	The value you wish to cast to is either string or binary type.
-------	--

### Usage

Converts a given string to a binary value. The function takes a string as its argument, and the function returns a binary value. Note that the function can also take a binary value as its argument, but it will return the same binary value.

### Examples

```
to_binary('0123456789abcdef') returns a binary value equivalent to 0x30313233343536373839616263646566
```

to\_binary('Hello there!') returns a binary value equivalent to 0x48656c6c6f20746865726521

to\_string( to\_binary('Good morning.')) returns the string 'Good morning.' after casting it to binary type and then back to string type.

### to\_bigdatetime()

Scalar. Converts a given value to a bigdatetime.

#### Syntax

```
to_bigdatetime ( value )
to_bigdatetime ( value, format )
```

value	A string, float, long, or bigdatetime. Strings must be in the format specified by the format argument. Numeric values represent the number of microseconds from the epoch (midnight, January 1, 1970 UTC).
format	A format string. Only valid if the value is a string. Must be one of the format codes for a bigdatetime. See "Date/Time Format Codes" for more information.

#### Parameters

### Usage

Converts a given value to a bigdatetime. The function takes a float, a long, or a string (and associated format string) as its argument, and the function returns a bigdatetime. Note that the function can also take a bigdatetime as its argument, but it will return the same bigdatetime.

### Examples

to\_bigdatetime(360000000) returns 1970-01-01 01:00:00.000000.

to\_bigdatetime('02/19/2010 10:15', '%m/%d/%Y %H:%M') returns 2010-02-19 10:15:00.000000.

to\_bigdatetime('07/19/2010 10:15 -07.00', 'MM/DD/YYYY HH:MI TZH:TZM') returns 2010-07-19 03:15:00.000000.

### to\_boolean()

Scalar. Converts a given value to a Boolean value.

#### Syntax

to\_boolean ( value )

#### Parameter

value	A string, or a Boolean value.
-------	-------------------------------

### Usage

Converts a given string to a Boolean value. The function takes a string as its argument, and the function returns a Boolean value. Note that the function can also take a Boolean value as its argument, but it will return the same Boolean value.

The strings "True", "Yes", and "On", regardless of case, or the numeral "1" returns TRUE. NULL returns NULL. Any other string returns FALSE.

#### Examples

to\_boolean ('1')returns TRUE.

to\_boolean ('FALSE')returns FALSE.

to\_boolean ('example')returns FALSE.

### to\_date()

Scalar. Converts a given value to a date.

#### Syntax

```
to_date ( value )
to_date ( value, format )
```

#### Parameters

value	A string, float, long, or date. Strings must be in the format specified by the format argument. Numeric values represent the number of microseconds from the epoch (midnight, January 1, 1970 UTC).
format	A format string. Only valid if the value is a string. Must be one of the format codes for a date. See "Date/Time Format Codes" for more information.

### Usage

Converts a given value to a date. The function takes a float, a long, or a string (and associated format string) as its argument, and the function returns a date. Note that the function can also take a date as its argument, but it will return the same date.

#### Examples

```
to_date('02/19/2010 10:15', '%m/%d/%Y %H:%M') returns 2010-02-19 10:15:00.
```

to\_date('07/19/2010 10:15 -07.00', 'MM/DD/YYYY HH:MI TZH:TZM') returns 2010-07-19 03:15:00.

### to\_float()

Scalar. Converts a given value to a float.

#### Syntax

to\_float ( value )

value	A string, interval, date/time type, numeric type, or money type.
-------	--

Converts a given value to a float. The function takes a string, interval, date/time type, numeric type, or money type as its argument, and the function returns a float . Note that the function can also take a float as its argument, but it will return the same float value.

A string converts based on the format for a float literal. An interval returns a value representing a number of microseconds. A date/time type returns a value representing the number of microseconds from the epoch (midnight, January 1, 1970 UTC). Those date/time types prior to the epoch convert to a negative value.

### Example

to\_float ('100.0')returns 100.0.

### to\_integer()

Scalar. Converts a given value to an integer.

#### Syntax

to\_integer ( value )

#### Parameters

value	The boolean, money, string, date, or any numeric type value you
	wish to cast to integer.

#### Usage

Converts a given value to an integer. The function takes a string, date, or any numeric type as its argument, and the function returns an integer. Note that the function can take an integer as its argument, but it will return the same integer.

Numeric values return the integer portion of the value. Values outside the valid range for an integer, or nonnumeric characters in a string value, return NULL. A date returns a value representing the number of seconds from the epoch (midnight, January 1, 1970 UTC). Those prior to the epoch convert to a negative value.

#### Example

to\_integer ('1')returns 1.

### to\_interval()

Scalar. Converts a given value to an interval.

#### Syntax

to\_interval ( value )

#### Parameters

value	A string, long, float, or interval representing a number in micro-
	seconds. Strings must follow the format for an interval literal.

#### Usage

Converts a given value to an interval. The function takes a string, a long, or a float as its argument, and the function returns an interval. Note that the function can also take an interval as its argument, but it will return the same interval.

#### Example

to\_interval( '1234' ) returns 1234.

### to\_long()

Scalar. Converts a given value to a long.

#### Syntax

to\_long ( value )

#### Parameters

value	A string, interval, date/time type, numeric type, or money type.
-------	--

#### Usage

Converts a given string to a long. The function takes a string, interval, date/time type, numeric type, or money type as its argument, and the function returns a long. Note that the function can take a long as its argument, but it will return the same long.

Numeric types return the integer portion of the value. Strings with nonnumeric characters, or with values outside the valid range for a long, return NULL. An interval returns a number of microseconds. A date/time type returns a value representing the number of microseconds from the epoch (midnight, January 1, 1970 UTC). Those prior to the epoch convert to a negative value.

#### Example

to\_long ('23')returns 23.

### to\_money()

Scalar. Converts a given value to the appropriate money type, based on a given scale.

#### Syntax

to\_money ( value, scale )

#### Parameters

value	A string, or a numeric type. The string must be all numeric, but can include a decimal point.
scale	An integer from 1 to 15.

### Usage

Converts a given value to a money type, based on the given scale. The function takes a string or a numeric type as its argument, and the function returns a money.

### Example

to\_money (12.361, 2)returns 12.36.

### to\_xml()

Scalar. Converts a given value to XML.

#### Syntax

to\_xml ( value )

#### Parameters

value	A string, or an XML type object.
-------	----------------------------------

### Usage

Converts a given value to XML. The function takes a string as its argument, and the function returns a string. The function can also take an XML type object as its argument, but it will return the same object. The function is the same as **xmlparse()**, but it can also handle an XML input.

### Example

```
xmlserialize ( to_xml ('<t/>') ) returns '<t/>'. The string gets converted to XML, then back into a string.
```

### tonetbinary()

Scalar. Converts an integer in host byte order to a 4 byte binary in network byte order.

#### Syntax

tonetbinary ( integer )

integer	An integer in host byte order.
---------	--------------------------------

Takes an integer in host byte order and converts it to a 4 byte binary in network byte order. Works for positive and negative values.

### Example

tonetbinary (1224164) returns 0012ADE4.

tonetbinary (-1224164) returns FFED521C.

### to\_string()

Scalar. Converts a given value to a string.

#### Syntax

```
to_string ( value [, format] [, timezone] )
```

#### Parameters

value	A value of any datatype.
format	(Optional) A format string. Only valid if the value is a date/time or numeric type.
timezone	(Optional) A time zone. Only valid if the value is a date/time type. If none is specified, the UTC time zone is used.

### Usage

Converts a given value to a string. The function can take any datatype as its argument, and the function returns a string. Note that the function can take a string as its argument, but it will return the same string. This function converts values as follows:

- For integers or longs, the user can include an optional format string to specify the format for the output string. The format string follows the ISO standard for fprintf. The default for integer expressions is '%d', while the default for long expressions is '%lld'.
- For a date/time type, the user can include to specify the format of the output string. The string must be a valid timestamp format code.
- The optional time zone argument can only be used with a date/time type. This string must be a valid time zone string. If no time zone is specified, UTC will be used. See "Time Zones" and "List of Time Zones" for more information.
- The function works the same as xmlserialize() when converting an XML value to a string
- For binary values, the returned string can contain unprintable characters because the function does a simple cast from binary to string rather than performing a conversion. To convert to a hex string representation of the binary value, use the hex\_string() function.

For a float value, the user can include an optional format string that specifies the format for the output of the floating point number as a string. The format string can include the following characters:

. or D	Returns a decimal point in the specified position. Only one decimal point can be specified, or the output will contain number signs instead of the values.
9	Replaced in the output by a single digit of the value. The value is returned with as many characters as there are 9s in the format string.
	If the value is positive, a leading space is included to the left of the value. If the value is negative, a leading minus sign is included to the left of the value.
	Excess 9s to the left of the decimal point are replaced with spaces, while excess 9s to the right of the decimal point are replaced with zeros. Insufficient 9s to the left of the decimal point returns number signs instead of the value, while insufficient 9s to the right of the decimal point result in rounding.
0	To the left of the decimal point, replaced in the output by a single digit of the value or a zero, if the value does not have a digit in the position of the zero. To the right of the decimal point, treated as a 9.
	If the value is positive, a leading space is included to the left of the value. If the value is negative, a leading minus sign is included to the left of the value.
EEEE	Returns the value in scientific notation. The output for this format always includes a single digit before the decimal. Combine with a decimal point and 9s to specify precision. 9s to the left of the decimal point are ignored.
	Must be placed at the end of the format string.
S	Returns a leading or trailing minus sign (-) or plus sign (+), depending on whether the value is positive or negative. Can only be placed at the beginning or end of the format string.
	Eliminates the usual single leading space, but not leading spaces as the result of excess 9s, zeros, or commas.
\$	Returns a leading dollar sign in front of the value. Can be placed anywhere in the format string.
	Returns a comma in the specified position. If there are no digits to the left of the comma, the comma is replaced with a space.
	The user can specify multiple commas, but cannot specify a comma as the first character in the format, or to the right of the decimal point.
FM	Strips spaces from the output.

### Examples

to\_string (45642) returns '45642'. to\_string (1234.567,'999') returns '####'. to\_string (1234.567,'9999D999') returns '1234.567'. to\_string (1234.567,'.99999999EEEE') returns '1.23456700E+03'.

### to\_timestamp()

Scalar. Converts a given value to a timestamp.

#### Syntax

```
to_timestamp ( value )
to_timestamp ( value, format )
```

#### Parameters

value	A string, float, or long. Strings must be in the format specified by the format argument. Numeric values represent the number of mi- croseconds from the epoch (midnight, January 1, 1970 UTC).
format	A format string. Only valid if the value is a string. Must be one of the format codes for a timestamp. See "Date/Time Format Codes" for more information.

### Usage

Converts a given value to a timestamp. The function takes a float, a long, or a string (and associated format string) as its argument, and the function returns a timestamp. Note that the function can also take a timestamp as its argument, but it will return the same timestamp.

#### Examples

to\_timestamp('02/19/2010 10:15', '%m/%d/%Y %H:%M') returns 2010-02-19 10:15:00.000.

```
to_timestamp('07/19/2010 10:15 -07.00', 'MM/DD/YYYY HH:MI TZH:TZM') returns 2010-07-19 03:15:00.000.
```

### usecToTime()

Scalar. Converts a given number of microseconds to a bigdatetime.

#### Syntax

```
usecToTime ( microseconds )
```

#### Parameters

microseconds	A long representing the number of microseconds since the epoch
	(midnight, January 1, 1970 UTC).

#### Usage

Converts a given number of microseconds to a bigdatetime. The function takes a long as its argument, and the function returns a bigdatetime.

### Example

usecToTime (3661000001) returns 1970-01-01 01:01:01.000001.

# **XML** Functions

There are special scalar functions which are designed to correctly handle XML data.

### xmlconcat()

Scalar. Concatenates a number of XML values into a single value.

#### Syntax

```
xmlconcat ( value, value [,value ...] )
```

#### Parameters

value	An XML value.
-------	---------------

#### Usage

Concatenates a number of XML values into a single value. The function takes at least two XML values, and the function returns an XML value.

### Example

```
xmlconcat ( xmlparse(stringCol), xmlparse('<t/>')))
```

### xmlelement()

Scalar. Creates a new XML data element, with attributes and XML expressions within it.

#### Syntax

```
xmlelement ( name, [xmlattributes (string AS name, ..., string AS
name),]
[ XML value, ..., XML value] )
```

#### Parameters

string	Attribute name/value pairs. For example: 'attrValue' AS attrName results in attrName = "attrValue" attribute created in the resulting XML element.
name	The name of the new element. Must adhere to naming conventions.
XML value	An XML value representing a child element.

### Usage

Creates a new XML data element, with attributes and XML expressions within it. The function takes . The function returns an XML value.

### Example

```
\label{eq:linear} \begin{array}{l} \texttt{xmlelement} \ (\texttt{top}, \ \texttt{xmlattributes}(\texttt{'data'} \ \texttt{as} \ \texttt{attrl}), \\ \texttt{xmlparse}(\texttt{'<t/>'}) \ \texttt{returns} \ \texttt{a} \ \texttt{new} \ XML \ \texttt{element} \ \texttt{called} \ \texttt{top}, \ \texttt{with} \ \texttt{a'data'} \ \texttt{attribute} \ \texttt{and} \ \texttt{<t/scale} \ \texttt{<t/scale} \ \texttt{called} \ \texttt{top}, \ \texttt{with} \ \texttt{a'data'} \ \texttt{attribute} \ \texttt{and} \ \texttt{<t/scale} \ \texttt{<t/scale} \ \texttt{attribute} \ \texttt{and} \ \texttt{<t/scale} \ \texttt{attribute} \ \texttt{at
```

### xmlparse()

Scalar. Converts a string into an XML value.

#### Syntax

```
xmlparse ( string )
```

#### Parameters

value	The XML value represented as a string.
-------	--

#### Usage

Converts a string into an XML value. The function takes a string as its argument, and returns an XML value. Since there is no XML data type, the value returned from this function can only be used as input to other functions expecting XML as input, such as **xmlserialize()**.

#### Example

xmlserialize ( xmlparse ('<t/>') ) returns '<t/>'. The string gets converted into an XML value, then back into a string.

### xmlserialize()

Scalar. Converts an XML value into a string.

#### Syntax

```
xmlserialize ( value )
```

#### Parameters

value	An XML value.
-------	---------------

#### Usage

Converts an XML value into a string. The function takes an XML value as its argument, and returns a string.

#### Example

```
xmlserialize ( xmlparse ('<t/>') ) returns '<t/>: The string gets converted into an XML value, then back into a string.
```

# **Date and Time Functions**

Date and time functions set time zone parameters, date format code preferences, and define calenders.

### business()

Scalar. Determines the next business day from a date value, based on a specified offset.

#### Syntax

business ( calendarfile, datevalue, offset )

#### Parameters

calendarfile	A string representing the file path for a calendar file.
datevalue	A date/time type.
offset	A negative or positive integer (should not be zero).

#### Usage

The function returns the same datatype as the **datevalue** argument.

The **offset** argument can be any negative or positive integer, but it cannot be zero. The function returns NULL if the offset is zero, and logs an error message. Negative integers return previous business days.

#### Example

business('/cals/us.cal',v.TradeTime, 1) returns the next business day within the calendar us.cal after the TradeTime date.

#### businessday()

Scalar. Determines if a date value falls on a business day (neither a weekend nor a holiday).

**Note:** This function is supported in mixed case. The Event Stream Processor considers **businessday()** and **businessDay()** the same function.

#### Syntax

businessday ( calendarfile, datevalue )

calendar	A string representing the file path for a calendar file
datevalue	A date/time type

The function returns 1 if the date falls on a business day (true), or 0 otherwise (false). The function returns an integer.

### Example

businessDay('/cals/us.cal',v.TradeTime) returns 1 if the date portion of v.TradeTime falls on a business day, and 0 otherwise.

### date()

Scalar. Converts a date value into an integer with the digits YYYYMMDD.

#### Syntax

```
date ( datevalue )
```

#### Parameters

datevalue A date	
------------------	--

### Usage

The function returns an integer.

### Example

date (undate ('1991-04-01 12:43:32')) returns 19910401.

### dateceiling()

Scalar. Computes a new date-time based on the provided date-time, multiple and date\_part arguments, with subordinate date\_parts set to zero. The result is then rounded up to the minimum date\_part multiple that is greater than or equal to the input timestamp.

#### Syntax

```
dateceiling ( date_part, expression [, multiple] )
```

date_part	Keyword that identifies the granularity desired. Valid keywords are identified below.
expression	Date-time expression containing the value to be evaluated.
multiple	Contains a multiple of date_parts to be used in the operation, which if supplied must be a nonzero positive integer value. If none is provided or it is NULL, the value is assumed to be 1.

Keyword	Keyword mean- ing	Multiples
yy or year	Year	Any positive integers
qq or quarter	Quarter	Any positive integers
mm or month	Month	Any positive integers
wk or week	Week	Any positive integers
dd or day	Day	Any positive integers.
hh or hour	Hour	1, 2, 3, 4, 6, 8, 12 and 24
mi or minute	Minute	1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, and 60
ss or second	Second	1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, and 60
ms or millisecond	Millisecond	1, 2, 4, 5, 8, 10, 20, 25, 40, 50, 100, 125, 200, 250, 500, and 1000

Valid Date Part Keywords and Multiples

This function determines the next largest date\_part value expressed in the timestamp, and zeros out all date\_parts of finer granularity than date\_part.

Date\_part is a keyword, expression is any expression that evaluates or can be implicitly converted to a datetime (or timestamp) datatype, and multiple is an integer containing the multiples of date\_parts to be used in performing the ceiling operation. For example, to establish a date ceiling based on 10 minute intervals, use MINUTE or MI for the date\_part, and 10 as the multiple.

Known errors:

- The server generates an invalid argument error if the value of the required arguments evaluate to NULL.
- The server generates an invalid argument error if the value of the multiple argument is not within range valid for the specified date\_part argument. As an example, have the value of multiple be less than 60 if date\_part mi is specified.

Standards and Compatibility Sybase extension.

#### Example

```
dateceiling( 'MINUTE', to_timestamp('2010-05-04T12:00:01.123',
'YYYY-MM-DDTHH24:MI:SS.FF'))
returns '2010-05-04 12:01:00.000'
```

### datefloor()

Scalar. Computes a new date-time based on the provided date-time, multiple and date\_part arguments, with subordinate date\_parts set to zero. The result is then rounded down to the maximum date\_part multiple that is less than or equal to the input timestamp.

#### Syntax

```
datefloor ( date_part, expression [, multiple] )
```

date_part	Keyword that identifies the granularity desired. Valid keywords are identified below.
expression	Date-time expression containing the value to be evaluated.
multiple	Contains a multiple of date_parts to be used in the operation, which if supplied must be a nonzero positive integer value. If none is provided or it is NULL, the value is assumed to be 1.

Keyword	Keyword mean- ing	Multiples
yy or year	Year	Any positive integers
qq or quarter	Quarter	Any positive integers
mm or month	Month	Any positive integers
wk or week	Week	Any positive integers
dd or day	Day	Any positive integers.
hh or hour	Hour	1, 2, 3, 4, 6, 8, 12 and 24
mi or minute	Minute	1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, and 60
ss or second	Second	1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, and 60
ms or millisecond	Millisecond	1, 2, 4, 5, 8, 10, 20, 25, 40, 50, 100, 125, 200, 250, 500, and 1000

This function zeros out all datetime values with a granularity finer than that specified by date\_part. Date\_part is a keyword, and expression is any expression that evaluates or can be implicitly converted to a datetime (or timestamp) datatype. Multiple is an integer that contains the multiples of date\_parts to be used in performing the floor operation. For example, to establish a date floor based on 10 minute intervals, use MINUTE or MI for date\_part, and 10 as the multiple.

### Known errors:

- The server generates an "invalid argument" error if the value of the required arguments evaluate to NULL.
- The server generates an "invalid argument" error if the value of the multiple argument is not within a range valid for the specified datepart argument. As an example, have the value of multiple be less than 60 if date\_part mi is specified.

### Standards and compatibility

Sybase extension.

### Example

```
datefloor( 'MINUTE', to_timestamp('2010-05-04T12:00:01.123', 'YYYY-
MM-DDTHH24:MI:SS.FF'))
returns '2010-05-04 12:00:00.000'
```

## datename()

Scalar. Converts a date value into a string.

### Syntax

```
datename ( datevalue )
```

### Parameters

datevalue	A date or bigdatetime.
-----------	------------------------

### Usage

Converts a date value to a string of the form 'YYYY-MM-DD'. The function takes a date or bigdatetime as its argument, and the function returns a string.

## Example

datename (undate ('2010-03-03 12:34:34')) returns '20100303'.

### datepart()

Scalar. Returns an integer representing a portion of a date.

#### Syntax

```
datepart ( portion, datevalue )
```

#### Parameters

portion	One of the following strings:
	<ul> <li>The year, if the string is yy or yyyy.</li> <li>The month, if the string is mm or m.</li> <li>The day of the year, if the string is dy or y.</li> <li>The day of the month, if the string is dd or d.</li> <li>The day of the week, if the string is dw.</li> <li>The hour, if the string is hh.</li> <li>The minute, if the string is mi or n.</li> <li>The second, if the string is so or s.</li> </ul>
datevalue	A date or bigdatetime.

#### Usage

Returns an integer representing a portion of a date. The portions that the function can return are the year, the month, the day of the year, the day of the month, the day of the week, the hour, the minute, or the second. The function takes a string as the **portion** argument, and a date or bigdatetime for the **datevalue** argument. The function returns an integer.

#### Example

datepart ( 'ss', undate ('2010-03-03 12:34:34')) returns 34.

### dateround()

Scalar. Computes a new date-time based on the provided date-time, multiple and date\_part arguments, with subordinate date\_parts set to zero. The result is then rounded to the value of a date\_part multiple that is nearest to the input timestamp.

#### Syntax

dateround ( date\_part, expression [, multiple] )

date_part	Keyword that identifies the granularity desired. Valid keywords are
	identified below.
expression	Date-time expression containing the value to be evaluated.
------------	---
multiple	Contains a multiple of date_parts to be used in the operation, which if supplied must be a nonzero positive integer value. If none is provided or it is NULL, the value is assumed to be 1.

Keyword	Keyword mean- ing	Multiples
yy or year	Year	Any positive integers
qq or quarter	Quarter	Any positive integers
mm or month	Month	Any positive integers
wk or week	Week	Any positive integers
dd or day	Day	Any positive integers.
hh or hour	Hour	1, 2, 3, 4, 6, 8, 12 and 24
mi or minute	Minute	1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, and 60
ss or second	Second	1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, and 60
ms or millisecond	Millisecond	1, 2, 4, 5, 8, 10, 20, 25, 40, 50, 100, 125, 200, 250, 500, and 1000

Valid Date Part Keywords and Multiples

# Usage

This function rounds the datetime value to the nearest date\_part or multiple of date\_part, and zeros out all date\_parts of finer granularity than date\_part or its multiple. For example, when rounding to the nearest hour, the minutes portion is determined, and if  $\geq 30$ , then the hour portion is incremented by 1, and the minutes and other subordinate date parts are zeros.

Date\_part is a keyword, expression is any expression that evaluates or can be implicitly converted to a datetime (or timestamp) datatype, and multiple is an integer containing the multiples of date\_parts to be used in performing the rounding operation. For example, to round to the nearest 10-minute increment, use MINUTE or MI for date\_part, and 10 as the multiple.

Known errors:

• The server generates an "invalid argument" error if the value of the required arguments evaluate to NULL.

• The server generates an "invalid argument" error if the value of the multiple argument is not within a range valid for the specified datepart argument. As an example, the value of multiple must be less than 60 if date\_part mi is specified.

### Example

```
dateround( 'MINUTE', to_timestamp('2010-05-04T12:00:01.123', 'YYYY-
MM-DDTHH24:MI:SS.FF'))
returns '2010-05-04 12:00:00.000'
```

# dayofmonth()

Scalar. Returns the integer representing the day of the month extracted from a given bigdatetime.

### Syntax

```
dayofmonth ( bigdatetime [ ,timezone ] )
```

### Parameters

bigdatetime	A bigdatetime value.
timezone	(Optional) A string representing a valid time zone. If none is specified, UTC is used. See "Time Zones" and "List of Time Zones" for more information.

# Usage

Returns an integer representing the day of the month extracted from a given bigdatetime. The function takes a bigdatetime as its argument (and an optional string representing a time zone), and the function returns an integer.

# Example

```
dayofmonth ((unbigdatetime ('2010-03-03 12:34:34:059111')) returns 3.
```

# dayofweek()

Scalar. Returns the integer representing the day of the week (Sunday is 1) extracted from a given bigdatetime.

# Syntax

dayofweek ( bigdatetime [ ,timezone ] )

#### Parameters

bigdatetime	A bigdatetime value.
-------------	----------------------

timezone	(Optional) A string representing a valid time zone. If none is
	specified, UTC is used. See "Time Zones" and "List of Time
	Zones" for more information.

### Usage

Returns an integer representing the day of the week extracted from a given bigdatetime. The function takes a bigdatetime as its argument (and an optional string representing a time zone), and the function returns an integer. Sunday is represented by 1, and the rest of the days of the week follow.

### Example

```
dayofweek ((unbigdatetime ('2010-03-03 12:34:34:059111')) returns 4.
```

# dayofyear()

Scalar. Returns the integer representing the day of the year extracted from a given bigdatetime.

#### Syntax

dayofyear ( bigdatetime [ ,timezone ] )

#### Parameters

bigdatetime	A bigdatetime value.
timezone	(Optional) A string representing a valid time zone. If none is specified, UTC is used. See "Time Zones" and "List of Time Zones" for more information.

### Usage

Returns an integer representing the day of the year extracted from a given bigdatetime. The function takes a bigdatetime as its argument (and an optional string representing a time zone), and the function returns an integer.

### Example

```
dayofyear ((unbigdatetime ('2010-03-03 12:34:34:059111')) returns 62.
```

# hour()

Scalar. Returns an integer representing the hour extracted from a given bigdatetime.

### Syntax

```
hour ( bigdatetime [ ,timezone ] )
```

Parameters

bigdatetime	A bigdatetime value.
timezone	(Optional) A string representing a valid time zone. If none is specified, UTC is used. See "Time Zones" and "List of Time Zones" for more information.

### Usage

Returns an integer representing the hour extracted from a given bigdatetime. The function takes a bigdatetime as its argument (and an optional string representing a time zone), and the function returns an integer.

# Example

```
hour ((unbigdatetime ('2010-03-03 12:34:34:059111')) returns 12.
```

# intdate()

Scalar. Converts an integer representing the number of seconds since 1970-01-01 00:00:00 UTC (the Epoch) to a date.

Note: This function is supported in mixed case. The Event Stream Processor supports both intdate() and intDate(), and considers them the same function.

# Syntax

intdate ( number )

### Parameters

number	An integer representing the number of seconds since 1970-01-01
	00:00:00 UTC (the Epoch).

### Usage

Converts a value representing the number of seconds since 1970-01-01 00:00:00 UTC (the Epoch) to a date. The function takes an integer as its argument, and the function returns a date.

# Example

intDate(1) returns a date, 1970-01-01 00:00:01.

# makebigdatetime()

Scalar. Constructs a bigdatetime from the given values.

# Syntax

```
makebigdatetime ( year, month, day, hour, minute, second, microsecond
[ ,timezone ] )
```

#### Parameters

year	An expression that evaluates to a value from 0001 to 9999. Values outside of the range 1970 to 2099 may result in inaccuracies due to leap years and daylight savings time.
month	An expression that evaluates to a value specifying the month. 0-12 indicate January to December, with both 0 and 1 representing January. Values larger than 12 roll over into subsequent years, while negative values subtract months from January of the specified year.
day	An expression that evaluates to a value specifying the day of the month. 0 and 1 both represent the first day of the year. Values larger than the valid number of days for the specified month roll over into subsequent months, while negative values subtract days from the first day of the specified month.
hour	An expression that evaluates to a value specifying the hour of the day. Values larger than 23 roll over into subsequent days, while negative values subtract hours from midnight of the specified day.
minute	An expression that evaluates to a value specifying the minute. Values larger than 59 roll over into subsequent hours, while nega- tive values subtract minutes from the specified hour.
second	An expression that evaluates to a value specifying the second. Values larger than 59 roll over into subsequent minutes, while negative values subtract seconds from the specified minute.
microsecond	An expression that evaluates to a value specifying the microsecond. Values larger than 999999 roll over into subsequent seconds, while negative values subtract microseconds from the specified second.
timezone	(Optional) A string representing the time zone. If omitted, the engine assumes the local time zone. See "Time Zones" and "List of Time Zones" for more information about valid time zone strings.

# Usage

Constructs a bigdatetime from the given values. The function takes integer values as its arguments (with the exception of the optional string representing a time zone), and the function returns an bigdatetime. If any argument is NULL, the function returns NULL.

### Example

to\_string (makebigdatetime (2010, 3, 3, 12, 34, 34, 59111)) returns '2010-03-03 12:34:34:059111'.

### microsecond()

Scalar. Returns an integer representing the microsecond extracted from a given bigdatetime.

#### Syntax

```
microsecond ( bigdatetime [ ,timezone ] )
```

#### Parameters

bigdatetime	A bigdatetime value.
timezone	(Optional) A string representing the time zone. If none is specified, UTC is used. See "Time Zones" and "List of Time Zones" for more information.

### Usage

Returns an integer representing the microsecond extracted from a given bigdatetime. The function takes a bigdatetime as its argument (and an optional string representing a time zone), and the function returns an integer.

### Example

```
microsecond ((unbigdatetime ('2010-03-03 12:34:34:059111')) returns 059111.
```

# minute()

Scalar. Returns an integer representing the minutes extracted from a given bigdatetime.

### Syntax

```
minute ( bigdatetime [ ,timezone ] )
```

### Parameters

bigdatetime	A bigdatetime value.
timezone	(Optional) A string representing a valid time zone. If none is specified, UTC is used. See "Time Zones" and "List of Time Zones" for more information.

# Usage

Returns an integer representing the minutes extracted from a given bigdatetime. The function takes a bigdatetime as its argument (and an optional string representing a time zone), and the function returns an integer.

### Example

```
minute ((unbigdatetime ('2010-03-03 12:34:34:059111')) returns 34.
```

### month()

Scalar. Returns an integer representing the month extracted from a given bigdatetime.

### Syntax

```
month ( bigdatetime [ ,timezone ] )
```

### Parameters

bigdatetime	A bigdatetime value.
timezone	(Optional) A string representing a valid time zone. If none is specified, UTC is used. See "Time Zones" and "List of Time Zones" for more information.

# Usage

Returns an integer representing the month extracted from a given bigdatetime. The function takes a bigdatetime as its argument (and an optional string representing a time zone), and the function returns an integer.

# Example

```
month ((unbigdatetime ('2010-03-03 12:34:34:059111')) returns 3.
```

# second()

Scalar. Returns an integer representing the seconds extracted from a given bigdatetime.

### Syntax

```
second ( bigdatetime [ ,timezone ] )
```

### Parameters

bigdatetime	A bigdatetime value.
timezone	(Optional) A string representing the time zone. If none is specified, UTC is used. See "Time Zones" and "List of Time Zones" for more information.

# Usage

Returns an integer representing the seconds extracted from a given bigdatetime. The function takes a bigdatetime as its argument (and an optional string representing a time zone), and the function returns an integer. If either argument is NULL, the function returns NULL.

# Example

```
second ((unbigdatetime ('2010-03-03 12:34:34:059111')) returns 34.
```

# sysdate()

Scalar. Returns the current system date as a date value.

### Syntax

sysdate ()

# Usage

Returns the current system date as a date value. The function has no arguments, and the function returns a date.

### Example

sysdate() on March 3, 2010, at 12:34:34 returns 2010-03-03 12:34:34.

# systimestamp()

Scalar. Returns the current system date as a timestamp value.

### Syntax

systimestamp ()

# Usage

Returns the current date, based on the Event Stream Processor server clock time, as a timestamp value. This date may differ from real time if the **clock** command in esp\_client was used to change the rate or time of the server clock. The function has no arguments, and the function returns a timestamp.

# Example

systimestamp() on March 3, 2010, at 12:34:34:059 returns 2010-03-03 12:34:34:059.

# unbigdatetime()

Scalar. Converts a given string into a bigdatetime value.

# Syntax

unbigdatetime ( string )

### Parameters

string	A string representing a bigdatetime value.
--------	--

# Usage

Converts a given string into a bigdatetime value. The function takes a string as its argument, and the function returns a bigdatetime.

# Example

unbigdatetime ('2003-06-14 13:15:00:232323') returns 2003-06-14 13:15:00:232323.

# undate()

Scalar. Converts a given string into a date value.

### Syntax

undate ( string )

### Parameters

string	A string representing a date value.
--------	-------------------------------------

# Usage

Converts a given string into a date value. The function takes a string as its argument, and the function returns a date.

### Example

undate ('2003-06-14 13:15:00') returns 2003-06-14 13:15:0.

# weekendday()

Scalar. Determines if a given date/time type falls on a weekend.

Note: This function is supported in mixed case. The Event Stream Processor supports both weekendday() and weekendDay(), and considers them the same function.

# Syntax

weekendday ( calendarfile, datevalue )

### Parameters

calendar	A string representing the file path for a calendar file.
datevalue	A date/time type.

# Usage

Determines if a date/time type value falls on a weekend. The function returns 1 if the date/time type falls on a weekend (true), or 0 otherwise (false). The function takes a string to represent the calendar path, and a date/time type as the **datevalue**. The function returns an integer.

# Example

weekendDay('/cals/us.cal',v.TradeTime) returns 1 if the date portion of v.TradeTime falls on a weekend, and 0 otherwise.

### year()

Scalar. Returns an integer representing the year extracted from a given bigdatetime.

#### Syntax

```
year ( bigdatetime [ ,timezone ] )
```

#### Parameters

bigdatetime	A bigdatetime value.
timezone	(Optional) A string representing the time zone. If none is specified, UTC is used. See "Time Zones" and "List of Time Zones" for more information.

### Usage

Returns an integer representing the year extracted from a given bigdatetime. The function takes a bigdatetime as its argument (and an optional string representing a time zone), and the function returns an integer.

### Example

year ((unbigdatetime ('2010-03-03 12:34:34:059111')) returns 2010.

# **Other Functions**

Reference list for all functions that are neither aggregate nor scalar type functions.

# cacheSize()

Returns the size of the current bucket in the event cache.

### Syntax

cacheSize (cacheName)

# Usage

Returns the size of the current bucket in the event cache. The function takes the argument of the name of the event cache variable. It returns an integer.

### Example

This example obtains the top 3 distinct prices per trading symbol. In order to accomplish this task, the example makes use of the getCache(), cacheSize() and deleteCache() functions.

```
CREATE SCHEMA TradesSchema (
Id integer,
TradeTime date,
Venue string,
```

```
Symbol string,
    Price float,
    Shares integer
CREATE INPUT WINDOW OTrades SCHEMA
TradesSchema PRIMARY KEY (Id)
CREATE FLEX flexOp
    IN QTrades
    OUT OUTPUT WINDOW QTradesStats SCHEMA TradesSchema PRIMARY
KEY(Symbol,Price)
    BEGIN
       DECLARE
          typedef [integer Id; | date TradeTime; string Venue;
             string Symbol; float Price;
             integer Shares] QTradesRecType;
         eventCache(QTrades[Symbol], manual, Price asc) tradesCache;
          typeof(QTrades) insertIntoCache( typeof(QTrades) qTrades )
             integer counter := 0;
             typeof (QTrades) rec;
             long cacheSz := cacheSize(tradesCache);
             while (counter < cacheSz) {
                rec := getCache( tradesCache, counter );
                if( round(rec.Price,2) = round(gTrades.Price,2) ) {
                   deleteCache(tradesCache, counter);
                   insertCache( tradesCache, qTrades );
                   return rec;
                   break;
                } else if( qTrades.Price < rec.Price) {</pre>
                   break;
                counter++;
            if(cacheSz < 3) {
               insertCache(tradesCache, gTrades);
               return gTrades;
            } else {
               rec := getCache(tradesCache, 0);
               deleteCache(tradesCache, 0);
               insertCache(tradesCache, gTrades);
               return rec;
            }
            return null;
      END;
      ON OTrades {
         keyCache( tradesCache, [Symbol=QTrades.Symbol; ] );
         typeof(QTrades) rec := insertIntoCache( QTrades );
         if(rec.Id) {
            if(rec.Id <> OTrades.Id) {
               output setOpcode(rec, delete);
```

```
output setOpcode(QTrades, upsert);
};
END;
```

# coalesce()

Other. Returns the first non-NULL expression from a list of expressions.

### Syntax

```
coalesce ( expression [,...] )
```

### Parameters

expression	All expressions must be of the same datatype.
------------	---

# Usage

Returns the first non-NULL expression from a list of expressions. The arguments can be of any datatype, but they must be all of the same datatype. The function returns the same datatype as its arguments.

# Example

```
coalesce (NULL, NULL, 'examplestring', 'teststring', NULL)
returns 'examplestring'.
```

# dateint()

Converts a date to an integer that represents the number of seconds since 1970-01-01 00:00:00 UTC (the Epoch).

Note: This function is supported in mixed case. The Event Stream Processor supports both dateint() and dateint(), and considers them the same function.

# Syntax

```
dateint ( datevalue )
```

# Parameters

datevalue	A date.
-----------	---------

# Usage

Converts a date to an integer that represents the number of seconds since 1970-01-01 00:00:00 UTC (the Epoch). The function takes a date as its argument, and the function returns an integer.

# Example

```
dateint (undate ('1970:01:01 00:01:01')) returns 61.
```

# deleteCache()

Deletes a row at a particular location (specified by index) in the event cache.

### Syntax

deleteCache (cacheName, index)

### Parameters

index	Row index in the event cache as an integer.
-------	---

# Usage

Deletes a row at a particular location (specifed by the index) in the event cache. This index is 0 based. The function takes an integer as its argument, and the function removes the row. The function does not produce an output. Specifying of an invalid index parameter will result in the generation of a bad record.

# Example

This example obtains the top 3 distinct prices per trading symbol. In order to accomplish this task, the example makes use of the getCache(), cacheSize() and deleteCache() functions.

```
CREATE SCHEMA TradesSchema (
    Id integer,
    TradeTime date,
    Venue string,
    Symbol string,
    Price float,
    Shares integer
CREATE INPUT WINDOW OTrades SCHEMA
TradesSchema PRIMARY KEY (Id)
CREATE FLEX flexOp
    IN OTrades
    OUT OUTPUT WINDOW QTradesStats SCHEMA TradesSchema PRIMARY
KEY(Symbol, Price)
    BEGIN
       DECLARE
          typedef [integer Id; | date TradeTime; string Venue;
             string Symbol; float Price;
             integer Shares] QTradesRecType;
         eventCache(QTrades[Symbol], manual, Price asc) tradesCache;
          typeof(QTrades) insertIntoCache( typeof(QTrades) qTrades )
          ł
             integer counter := 0;
             typeof (QTrades) rec;
             long cacheSz := cacheSize(tradesCache);
             while (counter < cacheSz) {</pre>
                rec := getCache( tradesCache, counter );
```

```
if( round(rec.Price,2) = round(gTrades.Price,2) ) {
               deleteCache(tradesCache, counter);
               insertCache( tradesCache, qTrades );
               return rec;
               break;
            } else if( gTrades.Price < rec.Price) {</pre>
               break;
            counter++;
        if(cacheSz < 3) {
           insertCache(tradesCache, gTrades);
           return gTrades;
        } else {
           rec := getCache(tradesCache, 0);
           deleteCache(tradesCache, 0);
           insertCache(tradesCache, gTrades);
           return rec;
        }
        return null;
  END;
  ON OTrades {
     keyCache( tradesCache, [Symbol=QTrades.Symbol; ] );
     typeof(QTrades) rec := insertIntoCache( QTrades );
     if(rec.Id)
        if(rec.Id <> QTrades.Id) {
           output setOpcode(rec, delete);
        output setOpcode(QTrades, upsert);
     }
 };
END;
```

# exp()

Returns the value of e (the base of the natural logarithm) raised to the power of a given number.

### Syntax

```
exp ( value )
```

# Parameters

value     A float.	
--------------------	--

# Usage

Returns the value of e (the base of the natural logarithm, 2.78128) raised to the power of a given number. If the argument is invalid, the server logs a floating-point exception error.

# Example

exp (2.0) returns 7.3890.

# firstnonnull()

Other. Returns the first non-NULL expression from a list of expressions.

### Syntax

```
firstnonnull ( expression [,...] )
```

### Parameters

expression	All expressions must be of the same datatype.
------------	---

# Usage

Returns the first non-NULL expression from a list of expressions. The function takes arguments of any datatype, but they must be all of the same datatype. The function returns the same datatype as its argument. This function behaves exactly like **coalesce()**.

# Example

firstnonnull (NULL, NULL, 'examplestring', 'teststring', NULL) returns 'examplestring'.

# get\*columnbyindex()

Returns the value of a column identified by an index.

# Syntax

```
getbinarycolumnbyindex ( record, colname )
getstringcolumnbyindex ( record, colname )
getlongcolumnbyindex ( record, colname )
getdatecolumnbyindex ( record, colname )
gettimestampcolumnbyindex ( record, colname )
getbigdatetimecolumnbyindex ( record, colname )
getbigdatetimecolumnbyindex ( record, colname )
getbooleancolumnbyindex ( record, colname )
getbooleancolumnbyindex ( record, colname )
getfloatcolumnbyindex ( record, colname )
```

# Parameters

name	The name of a stream or window.
colindex	Integer corresponding to an index value of a column. Index is 0 based.

# Usage

Returns the value of a column identified by an index. The function takes a string for the**name** argument and an integer for the **colindex** argument. The function returns the same datatype as specified in the function's name (a string for **getstringcolumnbyindex()**, for example).

If **colname** argument evaluates to NULL or the specified column does not exist in the associated window or stream, the function returns NULL and generates an error message.

### Example

CREATE MEMORY STORE "memstore"; CREATE INPUT WINDOW iwin1 SCHEMA (a int, b string) PRIMARY KEY (a) MEMORY STORE "memstore";

If you assume that the input passed into iwin1 was (1, 'hello'), then getstringcolumnbyindex ( iwin1, 1 ) would return 'hello'.

# get\*columnbyname()

Returns the value of a column identified by an expression evaluated at runtime.

# Syntax

```
getbinarycolumnbyname ( name, colname )
getstringcolumnbyname ( name, colname )
getlongcolumnbyname ( name, colname )
getintegercolumnbyname ( name, colname )
getdatecolumnbyname ( name, colname )
gettimestampcolumnbyname ( name, colname )
getbigdatetimecolumnbyname ( name, colname )
getintervalcolumnbyname ( name, colname )
getbooleancolumnbyname ( name, colname )
```

### Parameters

name	The name of a stream or window.
colname	An expression that evaluates to the name of a column with the same datatype as the function, in the stream or window.
	The <b>colname</b> argument for <b>getstringcolumnbyname()</b> would have a string, for example.

# Usage

Returns the value of a column identified by an expression evaluated at runtime. The function takes a string for the **name**. The datatype of the **colname** arguments corresponds to the function type, such as a string for **getstringcolumnbyname()**. The function returns the same datatype as **colname** (as specified in the function's name).

If **colname** argument evaluates to NULL or the specified column does not exist in the associated window or stream, the function returns NULL and generates an error message.

### Example

```
CREATE MEMORY STORE "memstore";
CREATE INPUT WINDOW iwin1 SCHEMA (a int, b string)
PRIMARY KEY (a) MEMORY STORE "memstore";
```

```
If you assume that the input passed into iwin1 was (1, 'hello'), then getstringcolumnbyname ( iwin1, a ) would return 'hello'.
```

# getCache()

Returns the row specified by a given index from the current bucket in the event cache.

#### Syntax

getCache (cacheName, index )

#### Parameters

cacheName	The name of the event cache.
index	Row index in the event cache as an integer.

### Usage

Returns the row specified by a given index from the current bucket in the event cache. This index is 0 based. The function takes the name of the event cache and an integer as its arguments, and returns a row from the event cache. Specifying an invalid index parameter generates a bad record.

# Example

This example obtains the top 3 distinct prices per trading symbol. In order to accomplish this task, the example makes use of the getCache(), cacheSize() and deleteCache() functions.

```
CREATE SCHEMA TradesSchema (
    Id integer,
    TradeTime date,
    Venue string,
    Symbol string,
    Price float,
    Shares integer
)
;
CREATE INPUT WINDOW QTrades SCHEMA
TradesSchema PRIMARY KEY (Id)
CREATE FLEX flexOp
    IN QTrades
    OUT OUTPUT WINDOW QTradesStats SCHEMA TradesSchema PRIMARY
KEY(Symbol, Price)
    BEGIN
       DECLARE
```

```
typedef [integer Id; | date TradeTime; string Venue;
         string Symbol; float Price;
         integer Shares] QTradesRecType;
     eventCache(QTrades[Symbol], manual, Price asc) tradesCache;
      typeof(OTrades) insertIntoCache( typeof(OTrades) gTrades )
      {
         integer counter := 0;
         typeof (QTrades) rec;
         long cacheSz := cacheSize(tradesCache);
         while (counter < cacheSz) {
            rec := getCache( tradesCache, counter );
            if( round(rec.Price,2) = round(gTrades.Price,2) ) {
               deleteCache(tradesCache, counter);
               insertCache( tradesCache, gTrades );
               return rec;
               break;
            } else if( qTrades.Price < rec.Price) {</pre>
               break;
            counter++;
        if(cacheSz < 3) {
           insertCache(tradesCache, qTrades);
           return gTrades;
        } else {
           rec := getCache(tradesCache, 0);
           deleteCache(tradesCache, 0);
           insertCache(tradesCache, gTrades);
           return rec;
        ļ
        return null;
  END;
  ON OTrades {
     keyCache( tradesCache, [Symbol=QTrades.Symbol; ] );
     typeof(QTrades) rec := insertIntoCache( QTrades );
     if(rec.Id)
        if(rec.Id <> QTrades.Id) {
           output setOpcode(rec, delete);
        output setOpcode(QTrades, upsert);
 };
END;
```

# getData()

This function takes a database query, gets rows from an external database table and returns them in a vector of records.

### Syntax

```
getData(vector, service, query, expr1, ... exprn)
```

vector	the name of the vector in which to return the selected records	
service	the name of the service to use to make the database query, a string	
query	a query for the database, a string	
expr	additional parameter to pass to the database along with the query, any of the basic datatypes (such as money, integer, string)	

### Parameters

# Usage

Specify the name of the vector in which to put the records returned by the function as the first argument. The function returns a vector with the name specified, containing the selected records.

Specify the service to use when querying the database as the second argument. The services that can be used to make the database queries are defined in the service.xml file. See the *Administrators Guide* for more information about this file and the services described in it.

Specify the query to make of the database as the third argument. The query can be in any database query language (such as SQL) as long as the appropriate service is defined in the service.xml file. Specify any additional parameters to pass to the database along with the query as subsequent arguments.

**Note:** The query statement must include placeholders, marked by a "?" character, for any additional parameters being passed.

# Example

getData(v, 'MyService', 'SELECT coll, col2 FROM myTable WHERE id= ?', 'myId'); gets records from a table named "myTable" using a service named "MyService", selects the first two columns of every row where the "id" is equal to the value of "myId" and returns them in a vector named "v".

# getmoneycolumnbyindex()

Returns the value of a column identified by an index.

### Syntax

getmoneycolumnbyindex ( name, colindex, scale )

#### Parameters

name	The name of a stream or window.	
colname	Integer corresponding to an index value of a column. Index is 0 based.	

scale An integer between 1 and 15.	
------------------------------------	--

#### Usage

Returns the value of a column identified by an index. The function takes a string for the **name** and integers for the **colindex** and **scale** arguments. The function returns a money type with the specified scale.

If **colname** argument evaluates to NULL or the specified column does not exist in the associated window or stream, the function returns NULL and generates an error message.

### Example

```
CREATE MEMORY STORE "memstore";
CREATE INPUT WINDOW iwin1 SCHEMA (a money(1), b money(3))
PRIMARY KEY (a) MEMORY STORE "memstore";
```

If you assume that the input passed into iwin1 was (1.2, 1.23), then getmoneycolumnbyindex (iwin1, 1, 3) would return 1.123.

# getmoneycolumnbyname()

Returns the value of a column identified by an expression evaluated at runtime as a money type.

#### Syntax

```
getmoneycolumnbyname ( name, colname, scale )
```

#### Parameters

name	The name of a stream or window.	
colname	An expression that evaluates to the name of a column with a money datatype, in the stream or window.	
scale	An integer between 1 and 15.	

#### Usage

Returns the value of a column identified by an expression evaluated at runtime. The function takes a string for the **name** and **colname** arguments and an integer to represent the scale of the money type. The function returns a money type with the specifed scale.

If **colname** argument evaluates to NULL or the specified column does not exist in the associated window or stream, the function returns NULL and generates an error message.

### Example

```
CREATE MEMORY STORE "memstore";
CREATE INPUT WINDOW iwin1 SCEHMA (a money(1), b money(3))
PRIMARY KEY (a) MEMORY STORE "memstore";
```

If you assume that the input passed into iwin1 was (1.2, 1.23), then getmoneycolumnbyname ( iwin1, b, 3 ) would return 1.123.

# getrowid()

Other. Returns the sequence number of a given row in the window.

### Syntax

getrowid ( row )

#### Parameters

row	A row in a window.
-----	--------------------

### Usage

Returns the sequence number of a given row in the window. The function takes a window ID as its argument, and the function returns the sequence number of the row in the window. This sequence number is known as the rowid, assigned uniquely as the rows get inserted.

# Example

CREATE MEMORY STORE "memstore";

CREATE INPUT WINDOW iwin1 SCHEMA (a money(1), b money(3)) PRIMARY KEY (a) MEMORY STORE "memstore";

```
CREATE INPUT WINDOW iwin2 SCHEMA (a money(1), b money(3)) PRIMARY KEY (a) MEMORY STORE "memstore";
```

For these windows, getrowid (iwin1) would return 0 and getrowid (iwin2) would return 1.

# now()

Returns the current system date as a bigdatetime value.

### Syntax

now ()

# Usage

Returns the current system date as a bigdatetime value. The function has no arguments, and the function returns a bigdatetime. This function works the same as **sysbigdatetime()**.

# Example

now() on March 3, 2010, at 12:34:34:059111 returns 2010-03-03 12:34:34:059111.

# rank()

Other. Returns the position of the row in the current group (only used in **GROUP HAVING** expression).

### Syntax

rank()

# Usage

Returns the position of the row in the current group, starting from position 0. This function is useful only in a **GROUP HAVING** expressions only. This function has no arguments, and the function returns an integer.

# Example

rank() > 3 returns 1 for the first three rows in a group and 0 for all other rows.

# recordDataToRecord

Converts the binary errorRecord value to a RECORD datatype value, based on the schema of the specified source stream.

# Syntax

recordDataToRecord (string sourceStreamName, binary errorRecord)

# Parameters

*sourceStreamName* is a string that provides the name of the stream from which the error record originated. To allow type checking of the return type, it must be an actual name, not a variable that carries the name. If this argument does not point to an existing stream, **recordDataToRecord** returns a NULL after setting an error flag to indicate that a bad argument has been specified.

*errorRecord* is a binary that provides the record that triggered the error. This should always be the errorRecord field of the error stream.

**Note:** Passing any arbitrary binary string or a mismatching schema (stream) name results in undefined behavior ranging from garbage in the record to crashing the server. The arguments to this built-in must be the *sourceStreamName* and *errorRecord* fields of the same error stream.

# recordDataToString

Converts the binary errorRecord value to string format.

# Syntax

recordDataToString (string sourceStreamName, binary errorRecord)

### Parameters

The *sourceStreamName* is a string that provides the name of the stream from which the error record originated. This should always be the sourceStreamName field of an error stream. Specifying the name of another stream (such as the error stream) can cause a fatal error due to a schema mismatch. If this argument doesn't point to an existing stream, **recordDataToString** returns a NULL after setting an error flag to indicate that a bad argument was specified.

The *errorRecord* is a binary that provides the record that triggered the error. This should always be the errorRecord field of the error stream and the schema should always match the record.

**Note:** Passing any arbitrary binary string or a mismatching schema (stream) name will result in undefined behavior: ranging from garbage in the record to crashing the server. The arguments to this built-in should always be the *sourceStreamName* and *errorRecord* fields of the same error stream.

# sind()

Returns the sine of a given value, expressed in degrees.

#### Syntax

```
sind ( value )
```

### Parameters

value	A float.
-------	----------

# Usage

Returns the sine of a given value, expressed in degrees. The function takes a float as its argument, and the function returns a float.

### Example

sind(45.0) returns 0.850903525.

# sysbigdatetime()

Returns the current system date as a bigdatetime value.

### Syntax

sysbigdatetime ()

### Usage

Returns the current system date as a bigdatetime value. The function has no arguments, and the function returns a bigdatetime. This function works the same as **now()**.

# Example

sysbigdatetime() on March 3, 2010, at 12:34:34:059111 returns 2010-03-03 12:34:34:059111.

# totimezone()

Converts a date from the given time zone to a specified time zone.

### Syntax

totimezone ( datevalue, fromzone, tozone )

### Parameters

datevalue	A date or bigdatetime.	
fromzone	A string representing a legal time zone.	
tozone	A string representing a legal time zone.	

# Usage

Converts a date from a given time zone to a new time zone. The first argument is the date being converted, the second argument is the original time zone, and the third argument is the new time zone. Time zone values are taken from the industry-standard TZ database. The first argument must be a date; the second and third arguments must be strings that represent legal time zones. The function returns a date.

# Example

totimezone(v.TradeTime, 'GMT', 'EDT') converts the time portion of each TradeTime from Greenwich Mean Time to Eastern Daylight Time.

# APPENDIX A List of Keywords

Reserved words in CCL that are case-insensitive. Keywords cannot be used as identifiers for any CCL objects.

adapter	age(s)	all	and	as	asc
attach	auto	begin	break	case	cast
connection	continue	count	create	day(s)	declare
deduced	default	delete	delta	desc	distinct
dumpfile	dynamic	else	end	eventCache	every
exit	external	false	fby	filter	first
flex	for	foreign	foreignJava	from	full
group	groups	having	hour(s)	hr	if
import	in	inherits	inner	input	insert
into	is	join	keep	key	last
language	left	library	like	load	local
log	max	memory	micros	microsecond(s)	millis
millisec- ond(s)	min	minute(s)	module	money	name
new	nostart	not	nth	null	on
or	order	out	outfile	output	parameter(s)
pattern	primary	properties	rank	records	retain
return	right	row(s)	safedelete	schema	sec
second(s)	select	set	setRange	slack	start
static	store(s)	stream	sum	sync	switch
then	times	to	top	transaction	true
type	typedef	typeof	union	update	upsert

A list of keywords present in CCL:

# APPENDIX A: List of Keywords

values	when	where	while	window	within
xmlattri- butes	xmlelement				

# See also

• *Case-Sensitivity* on page 33

# APPENDIX B Date and Time Programming

Set time zone parameters, date format code preferences, and define calendars.

# **Time Zones**

A time zone is a geographic area that has adopted the same standard time, usually referred to as the local time.

Most adjacent time zones are one hour apart. By convention, all time zones compute their local time as an offset from GMT/UTC. GMT (Greenwich Mean Time) is an historical term, originally referring to mean solar time at the Royal Greenwich Observatory in Britain. GMT has been replaced by UTC (Coordinated Universal Time), which is based on atomic clocks. For all Sybase Event Stream Processor purposes, GMT and UTC are equivalent. Due to political and geographical practicalities, time zone characteristics may change over time. For example, the start date and end date of daylight saving time may change, or new time zones may be introduced in newly created countries.

Internally, Event Stream Processor always stores date and time type information as a number of seconds, milliseconds, or microseconds since midnight January 1, 1970 UTC, depending on the datatype. If a time zone designator is not used, UTC time is applied.

# Daylight Saving Time

Daylight saving time is considered if the time zone uses daylight saving time and if the specified timestamp is in the time period covered by daylight savings time. The starting and ending dates for daylight saving time are stored in a C++ library.

If the user specifies a particular time zone, and if that time zone uses daylight saving time, Event Stream Processor takes these dates into account to adjust the date and time datatype. For example, since Pacific Standard Time (PST) is in daylight saving time setting, the engine adjusts the timestamp accordingly:

```
to_timestamp('2002-06-18 13:52:00.123456 PST','YYYY-MM-DD
HH24:MI:SS.ff TZD')
```

# Transitioning from Standard Time to Daylight Savings Time and Vice-Versa

During the transition to and from daylight saving time, certain times do not exist. For example, in the US, during the transition from standard time to daylight savings time, the clock changes from 01:59 to 03:00; therefore 02:00 does not exist. Conversely, during the transition from daylight saving time to standard time, 01:00 to 01:59 appears twice during one night because the time changes from 2:00 to 1:00 when daylight saving time ends.

However, since there may be incoming data input during these undefined times, the engine must deal with them in some manner. During the transition to daylight savings time, Event Stream Processor interprets 02:59 PST as 01:59 PST. When transitioning back to standard time, Event Stream Processor interprets 02:00 PDT as 01:00 PST.

# **Changes to Time Zone Defaults**

If you do not specify a value for the optional time zone parameter in certain date and time functions, Event Stream Processor uses Coordinated Universal Time (UTC).

Corresponding functions in Sybase CEP defaulted to the server's local time zone when no parameter was specified. If you are migrating CEP projects that do not have a time zone defined, they will use UTC when converted to Event Stream Processor. To continue using the server's local time zone, explicitly set that time zone in the time zone parameter for the following functions:

Sybase CEP Functions	Event Stream Processor Functions
dayofmonth	dayofmonth
dayofweek	dayofweek
dayofyear	dayofyear
hour	hour
maketimestamp	makebigdatetime
microsecond	microsecond
minute	minute
month	month
second	second
to_string	to_string
year	year

# List of Time Zones

Event Stream Processor supports standard time zones and their abbreviations.

Below is a list of time zones used in the Event Stream Processor from the industry-standard Olson time zone (also known as TZ) database.

ACT	AET	AGT
ART	AST	Africa/Abidjan

Africa/Accra	Africa/Addis_Ababa	Africa/Algiers	
Africa/Asmera	Africa/Bamako	Africa/Bangui	
Africa/Banjul	Africa/Bissau	Africa/Blantyre	
Africa/Brazzaville	Africa/Bujumbura	Africa/Cairo	
Africa/Casablanca	Africa/Ceuta	Africa/Conakry	
Africa/Dakar	Africa/Dar_es_Salaam	Africa/Djibouti	
Africa/Douala	Africa/El_Aaiun	Africa/Freetown	
Africa/Gaborone	Africa/Harare	Africa/Johannesburg	
Africa/Kampala	Africa/Khartoum	Africa/Kigali	
Africa/Kinshasa	Africa/Lagos	Africa/Libreville	
Africa/Lome	Africa/Luanda	Africa/Lubumbashi	
Africa/Lusaka	Africa/Malabo	Africa/Maputo	
Africa/Maseru	Africa/Mbabane	Africa/Mogadishu	
Africa/Monrovia	Africa/Nairobi	Africa/Ndjamena	
Africa/Niamey	Africa/Nouakchott	Africa/Ouagadougou	
Africa/Porto-Novo	Africa/Sao_Tome	Africa/Timbuktu	
Africa/Tripoli	Africa/Tunis	Africa/Windhoek	
America/Adak	America/Anchorage	America/Anguilla	
America/Antigua	America/Araguaina	America/Argentina/Bue- nos_Aires	
America/Argentina/Cata- marca	America/Argentina/ComodRiva- davia	America/Argentina/Cordoba	
America/Argentina/Jujuy	America/Argentina/La_Rioja	America/Argentina/Mendoza	
America/Argentina/ Rio_Gallegos	America/Argentina/San_Juan	America/Argentina/Tucuman	
America/Argentina/Ush- uaia	America/Aruba	America/Asuncion	
America/Atka	America/Bahia	America/Barbados	
America/Belem	America/Belize	America/Boa_Vista	

America/Bogota	America/Boise	America/Buenos_Aires	
America/Cambridge_Bay	America/Campo_Grande	America/Cancun	
America/Caracas	America/Catamarca	America/Cayenne	
America/Cayman	America/Chicago	America/Chihuahua	
America/Coral_Harbour	America/Cordoba	America/Costa_Rica	
America/Cuiaba	America/Curacao	America/Danmarkshavn	
America/Dawson	America/Dawson_Creek	America/Denver	
America/Detroit	America/Dominica	America/Edmonton	
America/Eirunepe	America/El_Salvador	America/Ensenada	
America/Fort_Wayne	America/Fortaleza	America/Glace_Bay	
America/Godthab	America/Goose_Bay	America/Grand_Turk	
America/Grenada	America/Guadeloupe	America/Guatemala	
America/Guayaquil	America/Guyana	America/Halifax	
America/Havana	America/Hermosillo	America/Indiana/Indianapolis	
America/Indiana/Knox	America/Indiana/Marengo	America/Indiana/Petersburg	
America/Indiana/Vevay	America/Indiana/Vincennes	America/Indianapolis	
America/Inuvik	America/Iqaluit	America/Jamaica	
America/Jujuy	America/Juneau	America/Kentucky/Louisville	
America/Kentucky/Monti- cello	America/Knox_IN	America/La_Paz	
America/Lima	America/Los_Angeles	America/Louisville	
America/Maceio	America/Managua	America/Manaus	
America/Martinique	America/Mazatlan	America/Mendoza	
America/Menominee	America/Merida	America/Mexico_City	
America/Miquelon	America/Moncton	America/Monterrey	
America/Montevideo	America/Montreal	America/Montserrat	
America/Nassau	America/New_York	America/Nipigon	
America/Nome	America/Noronha	America/North_Dakota/Center	

America/Panama	America/Pangnirtung	America/Paramaribo	
America/Phoenix	America/Port-au-Prince	America/Port_of_Spain	
America/Porto_Acre	America/Porto_Velho	America/Puerto_Rico	
America/Rainy_River	America/Rankin_Inlet	America/Recife	
America/Regina	America/Rio_Branco	America/Rosario	
America/Santiago	America/Santo_Domingo	America/Sao_Paulo	
America/Scoresbysund	America/Shiprock	America/St_Johns	
America/St_Kitts	America/St_Lucia	America/St_Thomas	
America/St_Vincent	America/Swift_Current	America/Tegucigalpa	
America/Thule	America/Thunder_Bay	America/Tijuana	
America/Toronto	America/Tortola	America/Vancouver	
America/Virgin	America/Whitehorse	America/Winnipeg	
America/Yakutat	America/Yellowknife	Antarctica/Casey	
Antarctica/Davis	Antarctica/DumontDUrville	Antarctica/Mawson	
Antarctica/McMurdo	Antarctica/Palmer	Antarctica/Rothera	
Antarctica/South_Pole	Antarctica/Syowa	Antarctica/Vostok	
Arctic/Longyearbyen	Asia/Aden	Asia/Almaty	
Asia/Amman	Asia/Anadyr	Asia/Aqtau	
Asia/Aqtobe	Asia/Ashgabat	Asia/Ashkhabad	
Asia/Baghdad	Asia/Bahrain	Asia/Baku	
Asia/Bangkok	Asia/Beirut	Asia/Bishkek	
Asia/Brunei	Asia/Calcutta	Asia/Choibalsan	
Asia/Chongqing	Asia/Chungking	Asia/Colombo	
Asia/Dacca	Asia/Damascus	Asia/Dhaka	
Asia/Dili	Asia/Dubai	Asia/Dushanbe	
Asia/Gaza	Asia/Harbin	Asia/Hong_Kong	
Asia/Hovd	Asia/Irkutsk	Asia/Istanbul	
Asia/Jakarta	Asia/Jayapura	Asia/Jerusalem	

Asia/Kabul	Asia/Kamchatka	Asia/Karachi	
Asia/Kashgar	Asia/Katmandu	Asia/Krasnoyarsk	
Asia/Kuala_Lumpur	Asia/Kuching	Asia/Kuwait	
Asia/Macao	Asia/Macau	Asia/Magadan	
Asia/Makassar	Asia/Manila	Asia/Muscat	
Asia/Nicosia	Asia/Novosibirsk	Asia/Omsk	
Asia/Oral	Asia/Phnom_Penh	Asia/Pontianak	
Asia/Pyongyang	Asia/Qatar	Asia/Qyzylorda	
Asia/Rangoon	Asia/Riyadh	Asia/Riyadh87	
Asia/Riyadh88	Asia/Riyadh89	Asia/Saigon	
Asia/Sakhalin	Asia/Samarkand	Asia/Seoul	
Asia/Shanghai	Asia/Singapore	Asia/Taipei	
Asia/Tashkent	Asia/Tbilisi	Asia/Tehran	
Asia/Tel_Aviv	Asia/Thimbu	Asia/Thimphu	
Asia/Tokyo	Asia/Ujung_Pandang	Asia/Ulaanbaatar	
Asia/Ulan_Bator	Asia/Urumqi	Asia/Vientiane	
Asia/Vladivostok	Asia/Yakutsk	Asia/Yekaterinburg	
Asia/Yerevan	Atlantic/Azores	Atlantic/Bermuda	
Atlantic/Canary	Atlantic/Cape_Verde	Atlantic/Faeroe	
Atlantic/Jan_Mayen	Atlantic/Madeira	Atlantic/Reykjavik	
Atlantic/South_Georgia	Atlantic/St_Helena	Atlantic/Stanley	
Australia/ACT	Australia/Adelaide	Australia/Brisbane	
Australia/Broken_Hill	Australia/Canberra	Australia/Currie	
Australia/Darwin	Australia/Hobart	Australia/LHI	
Australia/Lindeman	Australia/Lord_Howe	Australia/Melbourne	
Australia/NSW	Australia/North	Australia/Perth	
Australia/Queensland	Australia/South	Australia/Sydney	
Australia/Tasmania	Australia/Victoria	Australia/West	

Australia/Yancowinna	BET	BST	
Brazil/Acre	Brazil/DeNoronha	Brazil/East	
Brazil/West	САТ	СЕТ	
CNT	CST	CST6CDT	
CTT	Canada/Atlantic	Canada/Central	
Canada/East-Saskatche- wan	Canada/Eastern	Canada/Mountain	
Canada/Newfoundland	Canada/Pacific	Canada/Saskatchewan	
Canada/Yukon	Chile/Continental	Chile/EasterIsland	
Cuba	EAT	ECT	
EET	EST	EST5EDT	
Egypt	Eire	Etc/GMT	
Etc/GMT+0	Etc/GMT+1	Etc/GMT+10	
Etc/GMT+11	Etc/GMT+12	Etc/GMT+2	
Etc/GMT+3	Etc/GMT+4	Etc/GMT+5	
Etc/GMT+6	Etc/GMT+7	Etc/GMT+8	
Etc/GMT+0	Etc/GMT-0	Etc/GMT-1	
Etc/GMT-10	Etc/GMT-11	Etc/GMT-12	
Etc/GMT-13	Etc/GMT-14	Etc/GMT-2	
Etc/GMT-3	Etc/GMT-4	Etc/GMT-5	
Etc/GMT-6	Etc/GMT-7	Etc/GMT-8	
Etc/GMT-9	Etc/GMT0	Etc/Greenwich	
Etc/UCT	Etc/UTC	Etc/Universal	
Etc/Zulu	Europe/Amsterdam	Europe/Andorra	
Europe/Athens	Europe/Belfast	Europe/Belgrade	
Europe/Berlin	Europe/Bratislava	Europe/Brussels	
Europe/Bucharest	Europe/Budapest	Europe/Chisinau	
Europe/Copenhagen	Europe/Dublin	Europe/Gibraltar	

Europe/Helsinki	Europe/Istanbul	Europe/Kaliningrad	
Europe/Kiev	Europe/Lisbon	Europe/Ljubljana	
Europe/London	Europe/Luxembourg	Europe/Madrid	
Europe/Malta	Europe/Mariehamn	Europe/Minsk	
Europe/Monaco	Europe/Moscow	Europe/Nicosia	
Europe/Oslo	Europe/Paris	Europe/Prague	
Europe/Riga	Europe/Rome	Europe/Samara	
Europe/San_Marino	Europe/Sarajevo	Europe/Simferopol	
Europe/Skopje	Europe/Sofia	Europe/Stockholm	
Europe/Tallinn	Europe/Tirane	Europe/Tiraspol	
Europe/Uzhgorod	Europe/Vaduz	Europe/Vatican	
Europe/Vienna	Europe/Vilnius	Europe/Warsaw	
Europe/Zagreb	Europe/Zaporozhye	Europe/Zurich	
Factory	GB	GB-Eire	
GMT	GMT+0	GMT-0	
GMT0	Greenwich	HST	
Hongkong	IET	IST	
Iceland	Indian/Antananarivo	Indian/Chagos	
Indian/Christmas	Indian/Cocos	Indian/Comoro	
Indian/Kerguelen	Indian/Mahe	Indian/Maldives	
Indian/Mauritius	Indian/Mayotte	Indian/Reunion	
Iran	Israel	JST	
Jamaica	Japan	Kwajalein	
Libya	MET	MIT	
MST	MST7MDT	Mexico/BajaNorte	
Mexico/BajaSur	Mexico/General	Mideast/Riyadh87	
Mideast/Riyadh88	Mideast/Riyadh89	NET	
NST	NZ	NZ-CHAT	

Navajo	PLT	PNT	
PRC	PRT	PST	
PST8PDT	Pacific/Apia	Pacific/Auckland	
Pacific/Chatham	Pacific/Easter	Pacific/Efate	
Pacific/Enderbury	Pacific/Fakaofo	Pacific/Fiji	
Pacific/Funafuti	Pacific/Galapagos	Pacific/Gambier	
Pacific/Guadalcanal	Pacific/Guam	Pacific/Honolulu	
Pacific/Johnston	Pacific/Kiritimati	Pacific/Kosrae	
Pacific/Kwajalein	Pacific/Majuro	Pacific/Marquesas	
Pacific/Midway	Pacific/Nauru	Pacific/Niue	
Pacific/Norfolk	Pacific/Noumea	Pacific/Pago_Pago	
Pacific/Palau	Pacific/Pitcairn	Pacific/Ponape	
Pacific/Port_Moresby	Pacific/Rarotonga	Pacific/Saipan	
Pacific/Samoa	Pacific/Tahiti	Pacific/Tarawa	
Pacific/Tongatapu	Pacific/Truk	Pacific/Wake	
Pacific/Wallis	Pacific/Yap	Poland	
Portugal	ROC	ROK	
SST	Singapore	SystemV/AST4	
SystemV/AST4ADT	SystemV/CST6	SystemV/CST6CDT	
SystemV/EST5	SystemV/EST5EDT	SystemV/HST10	
SystemV/MST7	SystemV/MST7MDT	SystemV/PST8	
SystemV/PST8PDT	SystemV/YST9	SystemV/YST9YDT	
Turkey	UCT	US/Alaska	
US/Aleutian	US/Arizona	US/Central	
US/East-Indiana	US/Eastern	US/Hawaii	
US/Indiana-Starke	US/Michigan	US/Mountain	
US/Pacific	US/Pacific-New	US/Samoa	
UTC	Universal	VST	

W-SU	WET	Zulu
------	-----	------

# **Date/Time Format Codes**

A list of valid components that can be used to specify the format of a date/time type: date, timestamp, or bigdatetime.

Date/time type formats must be specified with either the Event Stream Processor formatting codes, or a subset of timestamp conversion codes provided by the C++ strftime() function. The are a number of different valid codes, however, A valid date/time type specification can contain no more than one occurrence of a code specifying a particular time unit (for example, a code specifying the year).

**Note:** All designations of year, month, day, hour, minute, or second can also read a fewer number of digits than is specified by the code. For example, DD reads both two-digit and one-digit day entries.

Column Code	Description	Input	Output
ММ	Month (01-12; JAN = 01).	Y	Y
YYYY	Four-digit year.	Y	Y
YYY	Last three digits of year.	Y	Y
YY	Last two digits of year.	Y	Y
Y	Last digit of year.	Y	Y
Q	Quarter of year (1, 2, 3, 4; JAN-MAR = 1).	N	Y
MON	Abbreviated name of month (JAN, FEB,, DEC).	Y	Y
MONTH	Name of month, padded with blanks to nine char- acters (JANUARY, FEBRUARY,, DECEMBER).	Y	Y
RM	Roman numeral month (1-XII; JAN = I).	Y	Y
WW	Week of year (1-53), where week 1 starts on the first day of the year and continues to the seventh day of the year.	N	Y
W	Week of month (1-5), where week 1 starts on the first day of the month and continues to the seventh day of the month.	N	Y
D	Day of week $(1-7; SUNDAY = 1)$ .	N	Y

Event Stream Processor Time Formatting Codes
Column Code	Description	Input	Output
DD	Day of month (1-31).	Y	Y
DDD	Day of year (1-366).		Y
DAY	Name of day (SUNDAY, MONDAY,, SATUR- DAY).	Y	Y
DY	Abbreviated name of day (SUN, MON,, SAT).	Y	Y
нн	Hour of day (1-12).	Y	Y
HH12	Hour of day (1-12).	Y	Y
HH24	Hour of day (0-23).	Y	Y
AM	Meridian indicator (AM/PM).	Y	Y
РМ	Meridian indicator (AM/PM).	Y	Y
MI	Minute (0-59).	Y	Y
SS	Second (0-59).	Y	Y
SSSSS	Seconds past midnight (0-86399).	Y	Y
SE	Seconds since epoch (January 1, 1970 UTC). This format can only be used by itself, with the FF for- mat, and/or with the time zone codes TZD, TZR, TZH and TZM.	Y	Y
MIC	Microseconds since epoch (January 1, 1970 UTC).	Y	Y
FF	Fractions of seconds (0-999999). When used in output, FF produces six digits for microseconds. FFFF produces twelve digits, repeating the six dig- its for microseconds twice. (In most circumstances, this is not the desired effect.) When used in input, FF collects all digits until a non-digit is detected, and then uses only the first six, discarding the rest.	Y	Y
FF[1-9]	Fractions of seconds. For output only, produces the specified number of digits, rounding or padding with trailing zeros as needed.	N	Y

Column Code	Description	Input	Output
MS	Milliseconds since epoch (January 1, 1970 UTC). When used for input, this format code can only be combined with FF (microseconds) and the time zone codes TZD, TZR, TZH, TZM. All other format code combinations generate errors. Furthermore, when MS is used with FF, the MS code must pre- cede the FF code: for example, MS.FF.	Y	Y
FM	Fill mode toggle: suppress zeros and blanks or not (default: not).	Y	Y
FX	Exact mode toggle: match case and punctuations exactly (default: not).	Y	Y
RR	Lets you store 20th century dates in the 21st century using only two digits.	Y	Ν
RRRR	Round year. Accepts either four-digit or two-digit input. If two-digit, provides the same return as RR.	Y	N
TZD	Abbreviated time zone designator such as PST.	Y	Y
ТΖН	Time zone hour displacement. For example, -5 in- dicates a time zone five hours earlier than GMT.	N	Y
TZM	Time zone hour and minute displacement. For ex- ample, -5:30 indicates a time zone that is five hours and 30 minutes earlier than GMT.	Ν	Y
TZR	Time zone region name. For example, US/Pacific for PST.	N	Y

#### Strftime() Timestamp Conversion Codes

Instead of using Event Stream Processor time formatting codes, output timestamp formats can be specified using a subset of the C++ strftime() function codes. The following rules apply:

- Any timestamp format specification that includes a percent sign (%) is considered a strftime() code.
- Strings can only include one type of formatting codes: the Event Stream Processor formatting codes, or the strftime() codes.
- Some strftime() codes are valid only on Microsoft Windows or only on UNIX-like operating systems. Different implementations of strftime() also include minor differences in code interpretation. To avoid errors, ensure that both the ESP Server and the ESP Studio are on the same platform, and are using compatible strftime() implementations. It is also essential to confirm that the provided codes meet the requirements for the platform.

- All time zones for formats specified with strftime() are assumed to be the local time zone.
- strftime() codes cannot be used to specify date/time type input, only date/time type output.

Strftime() Code	Description	
%a	Abbreviated weekday name; example: "Mon".	
%A	Full weekday name: for example "Monday".	
%b	Abbreviated month name: for example: "Feb".	
%B	Full month name: for example "February".	
%c	Full date and time string: the output format for this code differs, depending on whether Microsoft Windows or a UNIX-like operating system is being used. Microsoft Windows output example: 08/26/08 20:00:00 UNIX-like operating system output example: Tue Aug 26 20:00:00 2008	
%d	Day of the month, represented as a two-digit decimal integer with a value between 01 and 31.	
%H	Hour, represented as a two-digit decimal integer with a value between 00 and 23.	
%I	Hour, represented as a two-digit decimal integer with a value between 01 and 12.	
%j	Day of the year, represented as a three-digit decimal integer with a value be- tween 001 and 366.	
%m	Month, represented as a two-digit decimal integer with a value between 01 and 12.	
%M	Minute, represented as a two-digit decimal integer with a value between 00 and 59.	
%p	Locale's equivalent of AM or PM.	
%S	Second, represented as a two-digit decimal integer with a value between 00 and 61.	
%U	Number of the week in the year, represented as a two-digit decimal integer with a value between 00 and 53, with Sunday considered the first day of the week.	
% W	Weekday number, represented as a one-digit decimal integer with a value be- tween 0 and 6, with Sunday represented as 0.	

The Event Stream Processor supports the following strftime() codes:

Strftime() Code	Description
%W	Number of the week in the year, represented as a two-digit decimal integer with a value between 00 and 53, with Monday considered the first day of the week.
%x	Full date string (no time): The output format for this code differs, depending on whether you are using Microsoft Windows or a UNIX-like operating system. Microsoft Windows output example: 08/26/08 UNIX-like operating system output example: Tue Aug 26 2008
%X	Full time string (no date).
%y	Year, without the century, represented as a two-digit decimal number with a value between 00 and 99.
%Y	Year, with the century, represented as a four-digit decimal number.
%%	Replaced by %.

# **Calendar Files**

A text file detailing the holidays and weekends in a given time period.

#### Syntax

```
weekendStart <integer>
weekendEnd <integer>
holiday yyyy-mm-dd
holiday yyyy-mm-dd
...
```

#### Components

weekendStart	An integer that represents a day of the week, when Monday=0, Tuesday=1,, Saturday=5, and Sun- day=6.
weekendEnd	An integer that represents a day of the week, when Monday=0, Tuesday=1,, Saturday=5, and Sun- day=6.
holiday	A day of the year, in the form yyyy-mm-dd. A calendar file can have unlimited holidays.

#### Usage

A calendar file is a text file that describes the start and end date of a weekend, and the holidays within the year. The lines beginning with '#' characters are ignored, and can be used to provide user clarification or comments.

Calendar files are loaded and cached on demand by the Event Stream Processor. If changes occur in any of the calendar files, a command must be sent to refresh the cached calendar data, the refresh\_calendars command.

#### Example

The following is an example of a legal calendar file:

# Sybase calendar data for US 1983 weekendStart 5 weekendEnd 6 holiday 1983-02-21 holiday 1983-04-01 holiday 1983-05-30 holiday 1983-07-04 holiday 1983-09-05 holiday 1983-11-24 holiday 1983-12-26

## A

acos() 149 ADAPTER START statement 61 adapters 19 custom 8 input adapters 19 output adapters 20 overview 7 aggregate functions 126 aggregates 126 any() 127 avg() 127 corr() 128 count() 130 count(distinct) 131 covar\_pop() 129 covar\_samp() 130 exp\_weighted\_avg() 131 first() 132 last() 133 lwm\_avg() 133 max() 134 meandeviation() 135 median() 135 min() 136 nth() 136 recent() 137 regr\_avgx() 137 regr\_avgy() 138 regr\_count() 139 regr\_intercept() 139 regr\_r2() 140 regr\_slope() 140 regr\_sxx() 141 regr\_sxy() 142 regr\_syy() 142 stddev\_pop() 143 stddev\_samp() 144 sum() 144 valueinserted() 145 var\_pop() 145 var\_samp() 146 vwap() 146 weighted\_avg() 147 xmlagg() 148

aggregation **GROUP BY clause 43 GROUP FILTER clause 43 GROUP ORDER BY clause 43** HAVING clause 43 AGING clause 87 ANSI syntax 40 any() 127 APIs supported languages 8 arccosine acos() 149 arcsine asin() 149 arctangent atan() 150 atan2() 150 AS clause 88 ascii() 180 asin() 149 atan() 150 atan2() 150 ATTACH ADAPTER statement 62 avg() 127 avgof() 151

## В

base64 binary() 181 base64\_string() 181 basic project components queries 101 bigdatetime format codes 240 binary functions base64\_binary() 181 base64\_string() 181 bitand() 151 bitclear() 152 bitflag() 152 bitflaglong() 153 bitmask() 153 bitmasklong() 154 bitnot() 154 bitor() 154bitset() 155

bitshiftleft() 155 bitshiftright() 156 bittest() 156 bittoggle() 157 bitxor() 157 concat() 183 extract() 184 fromnetbinary() 184 hex binary() 185 hex\_string() 185 length() 163 tonetbinary() 193 bitand() 151 bitclear() 152 bitflag() 152 bitflaglong() 153 bitmask() 153 bitmasklong() 154 bitnot() 154 bitor() 154 bitset() 155 bitshiftleft() 155 bitshiftright() 156 bittest() 156 bittoggle() 157 bitwise functions bitand() 151 bitclear() 152 bitflag() 152 bitflaglong() 153 bitmask() 153 bitmasklong() 154 bitnot() 154 bitor() 154 bitset() 155 bitshiftleft() 155 bitshiftright() 156 bittest() 156 bittoggle() 157 bitxor() 157 bitxor() 157 business() 199 businessday() 199

## С

cacheSize() 214 calendar 244 calendar functions 244 business() 199

businessday() 199 weekendday() 213 CASE clause 89 case-insensitive 33 case-sensitive 33 cast() 182 cbrt() 158 CCL advanced techniques overview 45 language components 23 order of elements 11 overview 9 CCL functions 119 CCL keywords 229 CCL statements reference 61 ceil() 158 char() 183 clause CASE 89 clauses AGING 87 AS 88 **FROM 103** FROM (ANSI syntax) 104 FROM (comma-separated syntax) 103 GROUP BY 43, 106 GROUP FILTER 43, 107 GROUP ORDER BY 43, 108 HAVING 43, 109 IN 90, 93, 94, 98 **KEEP 91** MATCHING 42, 110 ON (join syntax) 110, 112 OUT 90, 93, 94, 98 PARAMETERS 90, 93, 94, 98 PRIMARY KEY 95 SCHEMA 97 SELECT 113 STORE 98 STORES 90, 93, 94, 98 UNION 36 WHERE 35, 116 coalesce() 216 column access functions get\*columnbyindex() 219 get\*columnbyname() 220 getbigdatetimecolumnbyindex() 219 getbigdatetimecolumnbyname() 220

getbinarycolumnbyindex() 219 getbinarycolumnbyname() 220 getbooleancolumnbyindex() 219 getbooleancolumnbyname() 220 getdatecolumnbyindex() 219 getdatecolumnbyname() 220 getfloatcolumnbyindex() 219 getfloatcolumnbyname() 220 getintegercolumnbyindex() 219 getintegercolumnbyname() 220 getintervalcolumnbyindex() 219 getintervalcolumnbyname() 220 getlongcolumnbyindex() 219 getlongcolumnbyname() 220 getmoneycolumnbyindex() 223 getmoneycolumnbyname() 224 getstringcolumnbyindex() 219 getstringcolumnbyname() 220 gettimestampcolumnbyindex() 219 gettimestampcolumnbyname() 220 column/window access functions cacheSize() 214 deleteCache() 217 get\*columnbyindex() 219 get\*columnbyname() 220 getbigdatetimecolumnbyindex() 219 getbigdatetimecolumnbyname() 220 getbinarycolumnbyindex() 219 getbinarycolumnbyname() 220 getbooleancolumnbyindex() 219 getbooleancolumnbyname() 220 getCache() 221 getdatecolumnbyindex() 219 getdatecolumnbyname() 220 getfloatcolumnbyindex() 219 getfloatcolumnbyname() 220 getintegercolumnbyindex() 219 getintegercolumnbyname() 220 getintervalcolumnbyindex() 219 getintervalcolumnbyname() 220 getlongcolumnbyindex() 219 getlongcolumnbyname() 220 getmoneycolumnbyindex() 223 getmoneycolumnbyname() 224 getrowid() 225 getstringcolumnbyindex() 219 getstringcolumnbyname() 220 gettimestampcolumnbyindex() 219 gettimestampcolumnbyname() 220

columns **BIGROWTIME 12** ROWID 12 **ROWTIME 12** combining queries 36 comma-separated syntax 42 compare() 159 complex join example 42 concat() 183 conversion functions cast() 182 date() 200 dateint() 216 datename() 203 int32() 171 intdate() 208 real() 174 string() 178 timeToMsec() 186 timeTosec() 187 timeToUsec() 187 to\_bigdatetime() 188 to binary() 188 to\_boolean() 189 to date() 190 to\_float() 190 to\_integer() 191 to interval() 191 to\_long() 192 to money() 192 to\_string() 194 to\_timestamp() 196 to xml() 193 totimezone() 228 unbigdatetime() 212 undate() 213 xmlparse() 198 xmlserialize() 198 correlation coefficient corr() 128 cos() 159 cosd() 160 cosh() 160 cosine cos() 159 cosd() 160 cosh() 160 count-based retention 15 count() 130

count(distinct) 131 covar\_pop() 129 covar\_samp() 130 CREATE DELTA STREAM statement 64 CREATE FLEX statement 67 CREATE LIBRARY statement 70 CREATE LOG STORE statement 71 CREATE MEMORY STORE statement 73 CREATE MODULE statement 74 CREATE SCHEMA statement 17, 76 CREATE STREAM statement 79 CREATE WINDOW statement 80 custom adapters overview 8

#### D

data aging AGING clause 87 data-flow programming example 3 introduction 3 databases compared to Sybase Event Stream Processor 2 datatypes supported datatypes in Event Stream Processor 23 date format codes 240 date and time functions totimezone() 228 date() 200 date/time format codes 240 date/time functions business() 199 businessday() 199 date() 200 dateceiling() 200 datefloor() 202 dateint() 216 datename() 203 datepart() 204 dateround() 204 dayofmonth() 206 dayofweek() 206 dayofyear() 207 hour() 207 intdate() 208 makebigdatetime() 208

microsecond() 210 minute() 210 month() 211msecToTime() 186 now() 225 second() 211 secToTime() 186 sysbigdatetime() 227 sysdate() 212 systimestamp() 212 timeToMsec() 186 timeToSec() 187 timeToUsec() 187 trunc() 179 unbigdatetime() 212 undate() 213 usecToTime() 196 weekendday() 213 year() 214 dateceiling() 200 datefloor() 202 dateint() 216 datename() 203 datepart() 204 dateround() 204 daylight saving time (DST) 231 dayofmonth() 206 dayofweek() 206 dayofyear() 207 declaration functions 77 parameters 77 typedefs 46, 77 variables 48, 77 declare blocks **DECLARE** statement 77 declaring 50 global 45 local 45 overview 45 **DECLARE** statement 77 declaring parameters parameters 47 deleteCache() 217 delta streams 11, 17, 64 dependency loops 18 distance() 160 distancesquared() 161 **DST 231** 

# Ε

error stream 66 error streams 57 event cache functions cacheSize() 214 deleteCache() 217 getCache() 221 getrowid() 225 Event Stream Processor components 6 event streams overview 2 events delete 6 examples 2 insert 6 update 6 examples ANSI svntax 40 comma-separated syntax 42 complex join 40, 42 complex stream-window join 40 merging data 36 pattern matching 42 schema discovery 76 schema inheritance 76 simple full outer join 40 simple inner join 40 simple left join 40 simple stream-window left join 40 using a UNION operator 36 exp\_weighted\_avg() 131 exp() 218 exponential functions exp() 218 power() 167 exponential moving average exp\_weighted\_avg() 131 expressions compound expressions 31 simple expressions 31 external data input and output adapters 7 extract() 184

## F

files calendar 244 filtering data 35 filters WHERE clause 116 first value() See first() first() 132, 133 firstnonnull() 219 flex operators 50 CREATE FLEX statement 67 flex stream 67 floor() 162 format codes bigdatetime 240 date 240 date/time 240 timestamp 240 FROM clause 103 ANSI syntax 104 comma-separated syntax 103 fromnetbinary() 184 functions acos() 149 aggregate functions 126 any() 127 ascii() 180 asin() 149 atan() 150 atan2() 150 avg() 127 avgof() 151 base64\_binary() 181 base64\_string() 181 bitand() 151 bitclear() 152 bitflag() 152 bitflaglong() 153 bitmask() 153 bitmasklong() 154 bitnot() 154 bitor() 154 bitset() 155 bitshiftleft() 155 bitshiftright() 156 bittest() 156 bittoggle() 157 bitxor() 157 built-in functions 119 business() 199

businessday() 199

C/C++ functions 120, 122 cacheSize() 214 cast() 182 cbrt() 158 ceil() 158 char() 183 coalesce() 216 compare() 159 concat() 183 corr() 128 cos() 159 cosd() 160 cosh() 160 count() 130 count(distinct) 131 covar pop() 129 covar\_samp() 130 date() 200 dateceiling() 200 datefloor() 202 dateint() 216 datename() 203 datepart() 204 dateround() 204 dayofmonth() 206 dayofweek() 206 dayofyear() 207 deleteCache() 217 distance() 160 distancesquared() 161 exp\_weighted\_avg() 131 exp() 218 external functions 119, 120, 122, 124 extract() 184 first() 132 firstnonnull() 219 floor() 162 fromnetbinary() 184 get\*columnbyindex() 219 get\*columnbyname() 220 getbigdatetimecolumnbyindex() 219 getbigdatetimecolumnbyname() 220 getbinarycolumnbyindex() 219 getbinarycolumnbyname() 220 getbooleancolumnbyindex() 219 getbooleancolumnbyname() 220 getCache() 221 getData 222 getdatecolumnbyindex() 219

getdatecolumnbyname() 220 getfloatcolumnbvindex() 219 getfloatcolumnbyname() 220 getintegercolumnbyindex() 219 getintegercolumnbyname() 220 getintervalcolumnbyindex() 219 getintervalcolumnbyname() 220 getlongcolumnbyindex() 219 getlongcolumnbyname() 220 getmoneycolumnbyindex() 223 getmoneycolumnbyname() 224 getrowid() 225 getstringcolumnbyindex() 219 getstringcolumnbyname() 220 gettimestampcolumnbyindex() 219 gettimestampcolumnbyname() 220 hex\_binary() 185 hex string() 185 hour() 207 int32() 171 intdate() 208 isnull() 162 Java functions 120, 124 last() 133 left() 171 length() 163 like() 172 ln() 163 log10() 164 log2() 164logx() 165 lower() 172ltrim() 173 lwm\_avg() 133 makebigdatetime() 208 max() 134 maxof() 165 meandeviation() 135 median() 135 microsecond() 210 min() 136 minof() 165 minute() 210 month() 211msecToTime() 186 nextval() 166 now() 225 nth() 136 other functions 214

patindex() 173 pi() 166 power() 167 random() 167 rank() 226 real() 174 recent() 137 regexp firstsearch() 175 regexp replace() 175 regexp\_search() 176 regr\_avgx() 137 regr\_avgy() 138 regr count() 139 regr\_intercept() 139 regr\_r2() 140 regr slope() 140 regr\_sxx() 141 regr\_sxy() 142 regr\_syy() 142 replace() 177 right() 177 round() 167 rtrim() 178 scalar functions 148 second() 211 secToTime() 186 sign() 168 sin() 168 sind() 227 sinh() 169 SPLASH functions 119 sqrt() 169 stddev pop() 143 stddev\_samp() 144 string() 178 substr() 178 sum() 144 sysbigdatetime() 227 sysdate() 212 systimestamp() 212 tan() 169, 170 tanh() 170 timeToMsec() 186 timeToSec() 187 timeToUsec() 187 to\_bigdatetime() 188 to\_binary() 188 to\_boolean() 189 to\_date() 190

to float() 190 to\_integer() 191 to\_interval() 191 to long() 192 to\_money() 192 to string() 194 to\_timestamp() 196 to xml() 193 tonetbinary() 193 totimezone() 228 trim() 179 trunc() 179 unbigdatetime() 212 undate() 213 upper() 180 usecToTime() 196 user-defined 119 user-defined functions 119, 120, 122, 124 valueinserted() 145 var\_pop() 145 var samp() 146 vwap() 146 weekendday() 213 weighted\_avg() 147 xmlagg() 148 xmlconcat() 197 xmlelement() 197 xmlparse() 198 xmlserialize() 198 year() 214

# G

get\*columnbyindex() 219 get\*columnbyname() 220 getbigdatetimecolumnbyindex() 219 getbigdatetimecolumnbyname() 220 getbinarycolumnbyindex() 219 getbinarycolumnbyname() 220 getbooleancolumnbyindex() 219 getbooleancolumnbyname() 220 getCache() 221 getData function 222 getdatecolumnbyindex() 219 getdatecolumnbyname() 220 getfloatcolumnbyindex() 219 getfloatcolumnbyname() 220 getintegercolumnbyindex() 219 getintegercolumnbyname() 220 getintervalcolumnbyindex() 219

getintervalcolumnbyname() 220 getlongcolumnbyindex() 219 getlongcolumnbyname() 220 getmoneycolumnbyindex() 223 getmoneycolumnbyname() 224 getrowid() 225 getstringcolumnbyindex() 219 getstringcolumnbyname() 220 gettimestampcolumnbyindex() 219 gettimestampcolumnbyname() 220 GROUP BY clause 106 rank() 226 **GROUP FILTER clause 107** rank() 226 group filtering function rank() 226 **GROUP ORDER BY clause 108** rank() 226 GUI authoring See visual authoring

## Н

HAVING clause 109 rank() 226 hex\_binary() 185 hex\_string() 185 hour() 207 hyperbolic cosine cosh() 160 hyperbolic sine sinh() 169 hyperbolic tangent tanh() 170

## I

implicit
columns 12
windows 15
IMPORT statement 83
importing
CCL files 83
function definitions 83
IMPORT statement 83
parameters 83
schema definitions 83
variables 83
IN clause 90

input 12 input adapters overview 7 int32() 171 intdate() 208 intervals values 26 isnull() 162

#### J

joins ANSI syntax 40 cardinality 37 complex join example 40, 42 complex stream-window join example 40 examples 37 key field rules 39 simple full outer join example 40 simple inner join example 40 simple left join example 40 simple stream-window left join example 40 types 37

## Κ

KEEP clause 91 retention policies 15 key field rules 39 keywords 229

#### L

last\_value() See last() last() 133 left() 171 length() 163 like() 172 linear regression functions regr\_avgx() 137 regr\_avgy() 138 regr\_count() 139 regr\_intercept() 139 regr\_r2() 140 regr\_slope() 140 regr\_sxx() 141 regr\_sxy() 142 regr\_syy 142

linearly weighted moving average lwm\_avg() 133 ln() 163 LOAD MODULE statement 84, 90, 93, 94, 98 local 12 log store CREATE LOG STORE statement 71 **CREATE MEMORY STORE statement 73** features 55 log store loops 18 optimization techniques 57 state after recovery 55 log stores **CREATE LOG STORE statement 71** log10() 164 log2() 164 logarithmic functions ln() 163 log10() 164 log2() 164logx() 165 logx() 165 lower() 172 ltrim() 173 lwm\_avg() 133

#### Μ

makebigdatetime() 208 MATCHING clause 42 max() 134 maxof() 165 mean dervivation meanderivation() 135 meandeviation() 135 median() 135 memory store 18 **CREATE MEMORY STORE statement 73** microsecond() 210 min() 136 minof() 165 minute() 210 modularity 52-54, 84, 90, 93, 94, 98 **CREATE MODULE statement 74** overview 51 module create 52-54, 74 load 52-54, 84 parameters 54 use 52-54

modules rules for 51 monitoring 59 month() 211 msecToTime() 186

## Ν

named schema 5, 17 naming 33 nextval() 166 now() 225 nth() 136

# 0

ON clause join syntax 110, 112 opcodes defined 6 insert, update, and delete events 6 operators arithmetic operators 27 comparison operators 27 LIKE operators 27 logical operators 27 string operators 27 UNION operator 114 other functions 214 OUT clause 93 output 12 output adapters overview 7 output expiry AGING clause 87 overview 9 Sybase Event Stream Processor 1

#### Ρ

parameters 47 in modules 54 initializing parameters at runtime 47 PARAMETERS clause 94 parameter binding 54 patindex() 173 pattern matching 42 performance count-based retention 15

SLACK value 15 persistence **CREATE LOG STORE statement 71 CREATE MEMORY STORE statement 73** log store 18, 55 pi() 166 population-based variance function var pop() 145 POSIX regular expression functions regexp\_firstsearch() 175 regexp replace() 175 regexp\_search() 176 power() 167 PRIMARY KEY clause 95 projects development task flow 21 introduction 4

# Q

aueries basic syntax 101 FROM clause 103 GROUP BY clause 43, 106 **GROUP FILTER clause 43, 107 GROUP ORDER BY clause 43, 108** HAVING clause 43, 109 **KEEP** clause 91 MATCHING clause 42, 110 ON clause 112 SELECT 113 UNION operator 36, 114 WHERE clause 35, 116 query construction aggregating data 35 combining queries 35 filtering data 35 joining multiple datasources 35 using pattern-matching rules 35

## R

random() 167 rank() 107, 226 real() 174 recent() 137 recordDataToRecord 226 recordDataToString 226 regexp\_firstsearch() 175

regexp replace() 175 regexp\_search() 176 regr\_avgx() 137 regr avgy() 138 regr\_count() 139 regr intercept() 139 regr\_r2() 140 regr slope() 140 regr sxx() 141 regr\_sxy() 142 regr syv() 142 regular expression functions regexp firstsearch() 175 regexp\_replace() 175 regexp\_search() 176 replace() 177 retention 91 count-based 15 time-based 15 retention policies description 15 right() 177 round() 167 rounding functions ceil() 158 floor() 162 round() 167 rtrim() 178

# S

sample-based variance function var samp() 146 scalar acos() 149 ascii() 180 asin() 149 atan() 150 atan2() 150 avgof() 151 base64\_binary() 181 base64\_string() 181 bitand() 151 bitclear() 152 bitflag() 152 bitflaglong() 153 bitmask() 153 bitmasklong() 154 bitnot() 154 bitor() 154

bitset() 155 bitshiftleft() 155 bitshiftright() 156 bittest() 156 bittoggle() 157 bitxor() 157 business() 199 businessday() 199 cast() 182 cbrt() 158 ceil() 158 char() 183 compare() 159 concat() 183 cos() 159 cosd() 160 cosh() 160 date() 200 dateceiling() 200 datefloor() 202 dateint() 216 datename() 203 datepart() 204 dateround() 204 dayofmonth() 206 dayofweek() 206 dayofyear() 207 distance() 160 distancesquared() 161 exp() 218 extract() 184 floor() 162 fromnetbinary() 184 hex\_binary() 185 hex string() 185 hour() 207 int32() 171 intdate() 208 isnull() 162 left() 171 length() 163 like() 172 ln() 163 log10() 164log2() 164logx() 165 lower() 172 ltrim() 173 makebigdatetime() 208 maxof() 165 microsecond() 210 minof() 165 minute() 210 month() 211 msecToTime() 186 nextval() 166 now() 225 patindex() 173 pi() 166 power() 167 random() 167 real() 174 regexp\_firstsearch() 175 regexp\_replace() 175 regexp search() 176 replace() 177 right() 177 round() 167 second() 211 secToTime() 186 sign() 168 sin() 168 sind() 227 sinh() 169 sqrt() 169 string() 178 substr() 178 sysbigdatetime() 227 sysdate() 212 systimestamp() 212 tan() 169, 170 tanh() 170 timeToMsec() 186 timeToSec() 187 timeToUsec() 187 to\_bigdatetime() 188 to\_binary() 188 to\_boolean() 189 to date() 190 to\_float() 190 to\_integer() 191 to\_interval() 191 to\_long() 192 to money() 192 to\_string() 194 to\_timestamp() 196 to\_xml() 193 tonetbinary() 193

totimezone() 228 trim() 179 trunc() 179 unbigdatetime() 212 undate() 213 usecToTime() 196 weekendday() 213 xmlconcat() 197 xmlelement() 197 xmlparse() 198 xmlserialize() 198 vear() 214 scalar functions 148 rtrim() 178 upper() 180 schema 17 overview 5 SCHEMA clause 17, 97 scope for modules 51 **SDKs** supported languages 8 second() 211 secToTime() 186 SELECT clause 113 set functions avgof() 151 coalesce() 216 firstnonnull() 219 maxof() 165 minof() 165 sign() 168 sin() 168 sind() 227 sine sin() 168 sind() 227 sinh() 169 SLACK count-based retention 15 performance 15 SPLASH overview 10 SPLASH functions declaring 119 sart() 169 standard deviation functions stddev\_pop() 143 stddev\_samp() 144

stateful elements 14 stateless elements delta stream 64 statements **ADAPTER START 61** ATTACH ADAPTER 62 **CREATE DELTA STREAM 64 CREATE FLEX 67 CREATE LIBRARY statement 70** CREATE LOG STORE 18, 71 **CREATE MEMORY STORE 18, 73** CREATE MODULE 52-54, 74 CREATE SCHEMA 76 **CREATE STREAM 79 CREATE WINDOW 80** DECLARE 77 IMPORT 52-54, 83 LOAD MODULE 52-54, 74, 84 stddev\_samp() 143, 144 stddev() See stddev\_samp() stddeviation() See stddev\_samp() STORE clause 98 stores log store 18, 55 memory store 18 STORES clause 98 streams 11, 12, 59 error 57, 66 input 12, 79 introduction 4 local 12, 79 output 12, 79 schema 5, 17 structure 5, 17 string functions ascii() 180 char() 183 int32() 171 left() 171 like() 172 lower() 172 ltrim() 173 patindex() 173 real() 174 regexp\_firstsearch() 175 regexp\_replace() 175 regexp\_search() 176

replace() 177 right() 177 rtrim() 178 substr() 178 to\_string() 194 trim() 179 unbigdatetime() 212 undate() 213 upper() 180 string() 178 Studio overview 8 substr() 178 sum() 144 sysbigdatetime() 227 sysdate() 212 systimestamp() 212

# Т

tan() 169, 170 tangent tan() 169, 170 tanh() 170 text authoring overview 8 time zones 231, 232 time-based retention 15 timestamp format codes 240 timeToMsec() 186 timeToSec() 187 timeToUsec() 187 to\_bigdatetime() 188 to\_binary() 188 to\_boolean() 189 to\_date() 190 to float() 190 to\_integer() 191 to\_interval() 191 to\_long() 192 to\_money() 192 to string() 194 to\_timestamp() 196 to\_xml() 193 tonetbinary() 193 totimezone() 228 trigonometric functions acos() 149 asin() 149

atan() 150 atan2() 150 cos() 159 cosd() 160 cosh() 160 sin() 168 sind() 227 sinh() 169 tan() 169, 170 tanh() 170 trim() 179 trunc() 179 type transformation int32() 171 type transformation functions cast() 182 date() 200 dateint() 216 datename() 203 intdate() 208 real() 174 timeToMsec() 186 timeToSec() 187 timeToUsec() 187 to\_bigdatetime() 188 to\_binary() 188 to\_boolean() 189 to date() 190 to\_float() 190 to\_integer() 191 to\_interval() 191 to\_long() 192 to money() 192 to\_string() 194 to timestamp() 196 to\_xml() 193 unbigdatetime() 212 undate() 213 xmlparse() 198 xmlserialize() 198 typedefs 46

# U

unbigdatetime() 212 undate() 213 UNION operator 36, 114 unions 36, 114 unnamed windows 15 upper() 180

usecToTime() 196

#### V

valueinserted() 145 var\_pop() 145 var\_samp() 146 variables 48 variance functions var\_pop() 145 var\_samp() 146 visual authoring overview 8 vwap() 146

#### W

weekendday() 213 weighted average functions exp\_weighted\_avg() 131 lwm\_avg() 133 vwap() 146 weighted\_avg() 147 weighted moving average weighted\_avg() 147 weighted\_avg() 147 WHERE clause 35, 116 window input 80 local 80 named 80 output 80 window access functions cacheSize() 214

deleteCache() 217 getCache() 221 getrowid() 225 windows 11, 12 implicit 15 input 12, 14 introduction 4 local 12, 14 named 14 output 12, 14 persistence 55 schema 5, 17 structure 5, 17 unnamed 14, 15

# X

XML functions xmlagg() 148 xmlconcat() 197 xmlelement() 197 xmlparse() 198 xmlserialize() 198 xmlagg() 148 xmlconcat() 197 xmlelement() 197 xmlparse() 198 xmlserialize() 198

# Y

year() 214