Programmers Reference

SAP Sybase Event Stream Processor 5.1 SP03

Contents

CHAPTER 1   **Introduction**

Data-Flow Programming

SAP® Sybase® Event Stream Processor uses data-flow programming for processing event streams.

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.

The sample project shown in the figure shows a simple example of this.

Each of the continuous queries in this simple example—the VWAP aggregate, the IndividualPositions join object, and the ValueByBook aggregate—is a type of derived stream, as its schema is derived from other inputs in the diagram, rather than originating directly from external sources. You can create derived streams in a diagram using the simple query elements provided in the Studio Visual editor, or by defining your own explicitly.

Figure 1: Data-Flow Programming - Simple Example
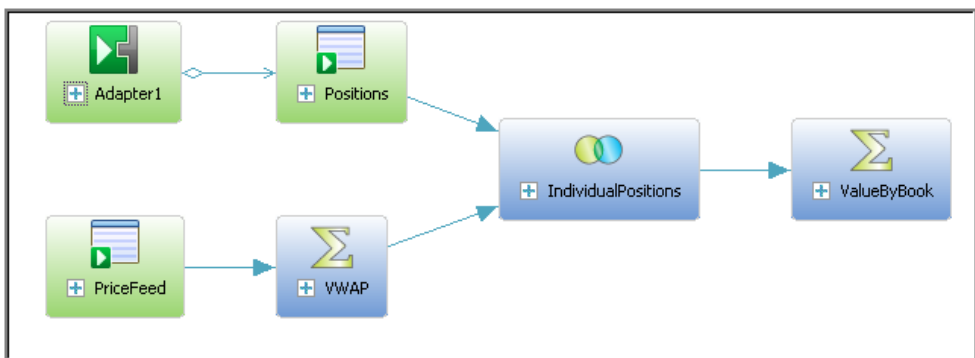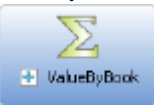
Table 1. Data-Flow Diagram Contents

Element	Description
PriceFeed	Represents an input window, where incoming data from an external source complies with a schema consisting of five columns, similar to a database table with columns. The difference is that in ESP, the streaming data is not stored in a database.
Positions	Another input window, with data from a different external source. Both Positions and PriceFeed are included as windows, rather than streams, so that the data can be aggregated.
VWAP	Represents a simple continuous query that performs an aggregation, similar to a SQL Select statement with a GROUP BY clause.
IndividualPositions	Represents a simple continuous query that performs a join of Positions and VWAP, similar to a SQL FROM clause that produces a join.
ValueByBook	Another simple query that aggregates data from the stream Individual Positions.

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 a 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.

Authoring Methods

SAP Sybase 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*.

CHAPTER 2 **CCL Project Basics**

ESP projects are written in CCL, a SQL-like language which specifies a data flow (by defining streams, windows, operations, and connections), and provides the capability to incorporate functions written in other languages, such as SPLASH, to handle more complex computational work.

Windows

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

You create a window if you need data to retain state. To create a Window, open the **Streams and Windows** compartment in the Visual editor in SAP Sybase Event Stream Processor Studio and click **Input Window**. When creating the window, and to retain rows, you must assign a primary key.

Since a window is a stateful element, with an underlying store, it can perform any operation specified by the opcode of an incoming event record. Depending on what changes are made to the contents of the store by the incoming event and its opcode, a window can produce output event records with different opcodes.

For example, if the window is performing aggregation logic, an incoming event record with an insert opcode can update the contents of the store and thus output an event record with an update opcode. The same could happen in a window implementing a left join.

A window can produce an output event record with same opcode as the input event record. If, for example, a window implemented a simple copy or a filter without any additional clauses, the input and output event records would have the same opcode.

An incoming event record with an insert opcode can produce an output event record with a delete opcode. For example, a window with a count-based retention policy (say keep 5 records) will delete those records from the store when the sixth event arrives, thus producing an output event record with a delete opcode.

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 a 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. Or, you can use the **ALL** modifier to explicitly specify that the window should retain all records.

Note: If you do not specify a retention policy, the window retains all records. This can be dangerous: the window can keep growing until all memory is used and the system shuts down. The only time you should have a window without a **KEEP** clause is if you know that the window size will be limited by incoming delete events.

Including the **EVERY** modifier in the **KEEP** clause produces a Jumping Window, which deletes all of the retained rows when the time interval expires or a row arrives that would exceed the maximum number of rows.

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 specify retention on input windows (or windows where data is copied directly from its source) using either log file-based stores or memory-based stores. For other windows, you can only specify retention on windows with memory-based stores

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.

Note: The SLACK value cannot be used with the EVERY modifier, and thus cannot be used in a Jumping Windows retention policy.

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 value s improve performance by reducing the need to constantly delete rows.

Count-based retention policies can also support retention based on content/column values using the PER sub-clause. A PER sub-clause can contain an individual column or a comma-delimited list of columns. A column can only be used once in a PER sub-clause. Specifying the primary key or autogenerate columns as a column in the PER sub-clause will result in a compiler warning. This is because these are unique entities for which multiple values cannot be retained.

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;
```

This example creates a Jumping Window named TotalCost from the source stream Trades. This window will retain a maximum of ten rows, and delete all ten retained rows on the arrival of a new row.

```
CREATE WINDOW TotalCost
PRIMARY KEY DEDUCTED
AS SELECT
                    trd.*,
                    trd.Price * trd.Size TotalCst
FROM Trades trd
KEEP EVERY 10 ROWS;
```

The following example creates a sliding window that retains 2 rows for each unique value of Symbol. Once 2 records have been stored for any unique Symbol value, arrival of a third record (with the same Symbol value) will result in deletion of the oldest stored record with the same Symbol value.

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

CREATE INPUT WINDOW TradesWin1
    SCHEMA TradesSchema
    PRIMARY KEY(Id)
    KEEP 2 ROWS PER(Symbol)
;
```

Time-based Retention
In a Sliding Windows time-based policy, a constant interval expression specifies the maximum age of the rows retained in the window. In a Jumping Window time-based retention

policy, all the rows produced in the specified time interval are deleted after the interval has expired.

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;
```

This example creates a Jumping Window named Win1 that keeps every row that arrives within the 100 second interval. When the time interval expires, all of the rows retained are deleted.

```
CREATE WINDOW Win1
PRIMARY KEY DEDUCED
AS SELECT * FROM Source1
KEEP EVERY 100 SECONDS;
```

The PER sub-clause supports content-based data retention, wherein data is retained for a specific time period (specified by an interval) for each unique column value/combination. A PER sub-clause can contain a single column or a comma-delimited list of columns, but you can use each column only once in the same PER clause.

Note: Time based windows retain data for a specified time regardless of their grouping.

The following example creates a jumping window that retains 5 seconds worth of data for each unique value of Symbol.

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

CREATE INPUT WINDOW TradesWin2
    SCHEMA TradesSchema
    PRIMARY KEY(Id)
    KEEP EVERY 5 SECONDS PER(Symbol)
;
```

Retention Semantics

When the insertion of one or more new rows into a window triggers deletion of preexisting rows (due to retention), the window propagates the inserted and deleted rows downstream to relevant streams and subscribers. However, the inserted rows are placed before the deleted rows, since the inserts trigger the deletes.

Aging Policy

An aging policy can be set to flag records that have not been updated within a defined interval. This is useful for detecting records that may be stale. Aging policies are an advanced, optional feature for a window or other stateful element.

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 or derived. Derived windows are either 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.

Table 2. Named Window Capabilities

Type	Receives Data From	Sends Data To	Visible Externally
input	Input adapter or external application that sends data into ESP using the ESP SDK	Other windows, delta streams, and/or output adapters	Yes
output	Other windows, streams, or delta streams	Other windows, delta streams, and/or output adapters	Yes
local	Other windows, streams, or delta streams	Other windows or delta streams	No

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.

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

An unnamed window is implicitly created when the **KEEP** clause is used with a source name in the **FROM** clause of a statement. On a Delta Stream, only unnamed windows can be created by specifying the **KEEP** clause in 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:

```
CREATE WINDOW MaxTradePrice
PRIMARY KEY DEDUCED
STORE S1
AS SELECT trd.Symbol, max(trd.Price) MaxPrice
FROM Trades trd KEEP 10 MINUTES
GROUP BY trd.Symbol;
```

This example creates a TotalCost Unnamed Window from the source stream Trades. Jumping Window will retain ten rows, and clear all rows on the arrival of the 11th row.

```
CREATE DELTA STREAM TotalCost
PRIMARY KEY DEDUCTED
AS SELECT
                  trd.*,
                  trd.Price * trd.Size TotalCst
FROM Trades trd
KEEP EVERY 10 ROWS;
```

In all three examples, `Trades` can be a delta stream, or a window.

Delta Streams

Delta streams are stateless elements that can understand all opcodes. Unlike streams, they are not limited to inserts and updates.

Delta streams can be classed as derived streams. Derived streams are either output or local. A delta stream is derived from an existing stream or window and is not an input stream. 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.

You must provide a primary key for delta streams. Delta streams are allowed key transformations only when performing aggregation, join, or flex operations. Because a delta stream does not maintain state, you cannot define a delta stream on a window where the keys differ.

While a delta stream does not maintain state, it can interpret all of the opcodes in incoming event records. The opcodes of output event records depend on the logic implemented by the delta stream.

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 ;
```

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 Capability	Streams	Windows	Delta Streams
Type of element	Stateless	Stateful, due to retention and store capabilities	Stateless
Data retention	None	Yes, rows (based on retention policy)	None
Available store types	Not applicable	Memory store or log store	Not applicable

Feature Capability	Streams	Windows	Delta Streams
Element types that can be derived from this element	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 deduced	Yes, explicit or deduced
Support for aggregation 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 **delete**	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.

Input/Output/Local

You can designate streams, windows, and delta streams as input or derived. Derived streams, including delta streams, are either 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 (Col1 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 (Col1 INTEGER, Col2 STRING);
CREATE MEMORY STORE myStore;
CREATE INPUT WINDOW IWin1 SCHEMA mySchema
    PRIMARY KEY(Col1)
    STORE myStore;
```

This is an output window:

```
CREATE SCHEMA mySchema (Col1 INTEGER, Col2 STRING);
CREATE MEMORY STORE myStore;
CREATE OUTPUT WINDOW OWin1
    PRIMARY KEY (Col1)
    STORE myStore
    AS SELECT  A.Col1 col1, 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 (Col1 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 (Col1 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;
```

Implicit Columns

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

Column	Datatype	Description
ROWID	long	Provides a unique row identification number for each row of incoming data.
ROWTIME	date	Provides the last modification time as a date with second precision.
BIGROWTIME	bigdatetime	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.

You can refer to these implicit columns just like any explicit column (for example, using the `stream.column` convention).

Event Block Lifecycle

An Event Block Lifecycle measures the interval between the creation of Event Blocks.

The two types of Event Blocks, Transactions and Envelopes, will carry on throughout a model as long as rows contained inside of the model are passing through the system.

A new Event Block is created each time data from a block is distributed to a new element in the model. If an element is feeding multiple destinations, such as streams, windows, and adapters, a separate transaction is sent to each element. Events can be filtered out of an Event Block without impacting the block.

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.

There are two ways to create a schema: you can create a named schema using the **CREATE SCHEMA** statement or you can create an inline schema within a stream or window definition.

Named schemas are useful when the same schema will be used in multiple places, since any number of streams and windows can reference a single named schema.

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
);
```

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 holds 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
The log store holds all data in memory, but also logs all data to the disk, meaning it guarantees data state recovery in the event of a failure. Use a log store to be able to recover the state of a window after a restart.

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 `logstore1` 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.

CCL Continuous Queries

Build a continuous 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 *SELECT Clause* for more information.
from_clause	Selects the source data is derived from. See below and *FROM Clause* for more information.
matching_clause	Used for pattern matching. See *MATCHING Clause* and *Pattern Matching* for more information.
where_clause	Performs a filter. See *WHERE Clause* and *Filters* for more information.
groupFilter_clause	Filters incoming data in aggregation. See *GROUP FILTER Clause* and *Aggregation* for more information.
groupBy_clause	Specifies what collection of rows to use the aggregation operation on. See *GROUP BY Clause* and *Aggregation* for more information.

`groupOrder_clause`	Orders the data in a group before aggregation. See *GROUP ORDER BY Clause* and *Aggregation* for more information.
`having_clause`	Filters data that is output by the derived components in aggregation. See *HAVING Clause* and *Aggregation* for more information.

Usage

CCL queries are embedded in the **CREATE STREAM**, **CREATE WINDOW**, and **CREATE DELTA STREAM** statements, and are applied to the inputs specified in the **FROM** clause of the query to define the contents of the new stream or window. The example below demonstrates the use of both the **SELECT** clause and the **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
[...]
```

Reference Table Queries

Reference Table Queries provide a way to augment the streaming data in an SAP Sybase Event Stream Processor project with information from a HANA table.

This CCL element enables you to establish a reference to a HANA table in an ESP project . This reference can then be used in a join along with streams and windows. When an event arrives via a stream or window, the reference executes a query on the HANA table and uses the returned data in the join to enrich streaming data with information from the database.

To create a reference you need the following information:

- the name of the database service to use
- the name of the table from which to retrieve information
- the schema of the table

You also have the option of specifying the following:

- the primary key of the table as the primary key of the reference
- that the reference should attempt to reconnect when the connection is lost
- how many attempts to make
- how long to wait between attempts

Use the **CREATE REFERENCE** statement to define the reference, and then use the **FROM** and **ON** clauses to join data from the reference with streams and windows in an ESP project.

Also, in SPLASH, you can use an iterator over a reference from a local **DECLARE** block and in the **FLEX** operator in the same way you use an iterator over a window. You may also iterate over a reference using the key search (if a primary key is defined), record matching search, and for loop functionality.

Adapters

Adapters connect the Event Stream Processor to the external world.

An input adapter connects an input stream or window to a data source. It reads the data output by the source and modifies it for use in an ESP project.

An output adapter connects an output stream or window to a data sink. It reads the data output by the ESP project and modifies it for use by the consuming application.

Adapters are attached to input streams and windows, and output streams and windows, using the **ATTACH ADAPTER** statement and they are started using the **ADAPTER START** statement. In some cases it may be important for a project to start adapters in a particular order. For example, it might be important to load reference data before attaching to a live event stream. Adapters can be assigned to groups and the **ADAPTER START** statement can control the start up sequence of the adapter groups.

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

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.

CHAPTER 3 **CCL Language Components**

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

Datatypes

SAP 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^{63} to 2^{63-1}). 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, with a size limit of 2 gigabytes.
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): -9223372036854775808 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 (-9...7) -1 to avoid CCL compiler errors.
	Specify explicit scale for money constants with **Dn** syntax, where **n** represents the scale. For example, 100.1234567D7, 100.12345D5.
	Implicit conversion between money(n) types is not supported because there is a risk of losing range or scale. Perform the **cast** 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: • All boolean, integer, and long values are converted in their original format to bigdatetime. • 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. • All date values are multiplied by 1000000 and converted to microseconds to satisfy bigdatetime format. • All timestamp values are multiplied by 1000 and converted to microseconds to satisfy bigdatetime format.
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 values that fit in a long when converted to microseconds are: • MICROSECONDS (MICROSECOND, MICROS): +/- 9223372036854775807 • MILLISECONDS (MILLISECOND, MILLIS): +/- 9223372036854775 • SECONDS(SECOND, SEC): +/- 9223372036854 • MINUTES(MINUTE, MIN): +/- 153722867280 • HOURS(HOUR,HR): +/- 2562047788 • DAYS(DAY): +/- 106751991 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.
binary	Represents a raw binary buffer. Maximum length of value is platform-dependent, with a size limit of 2 gigabytes. 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.

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.

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

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

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 MINUTES
```

```
2 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.

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 Preincrement (++*argument*) value is incremented before it is passed as an argument Postincrement (*argument*++) value is passed and then incremented	++a (preincrement) a++ (postincrement)
--	Decrement Predecrement (--*argument*) value is decremented before it is passed as an argument Postdecrement (*argument*--) value is passed and then decremented	--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 Usage
=	Equality	a0=a1
!=	Inequality	a0!=a1
<>	Inequality	a0<>a1
>	Greater than	a0!>a1
>=	Greater than or equal to	a0!>=a1

Operator	Meaning	Example Usage
<	Less than	a0!<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. **Note:** The + operator does not support mixed datatypes (such as an integer and a string).	'go' + 'cart'

LIKE Operator

May be used in column expressions and **WHERE** clause expressions. Use the LIKE operator to match string expressions to strings that closely resemble each other but do not exactly match.

Operator	Syntax and Meaning	Example Usage

LIKE	Matches **WHERE** clause string expressions to strings that closely resemble each other but do not exactly match.
	
	`compare_expression LIKE pat-`
	`tern_match_expression`
	
	The LIKE operator returns a value of TRUE if **compare_expression** matches **pattern_match_expression**, or FALSE if it does not. The expressions can contain wildcards, where the percent sign (%) matches any length string, and the underscore (_) matches any single character.
		Trades.StockName LIKE "%Corp%"

[] 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 current row in a stream or window. `stream-or-window-name[index].column` **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.	MyNamedWindow[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 ^ operator is right-associative. Thus, a ^ b ^ c = a ^ (b ^ c), not (a ^ b) ^ 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()`

Sequences of Expressions

An expression can contain a sequence of expressions; separated by semicolons and grouped using parentheses, to be evaluated in order. The type and value of the expression is the type and value of the last expression in the sequence. For example,

- `(var1 := v.Price; var2 := v.Quantity; 0.0)`

sets the values of the variables var1 and var2, and then returns the value `0.0`.

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
```

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.

The visual editor in the SAP Sybase Event Stream Processor Studio recognizes a doc-comment and puts it in the comment field of the top-level CCL statement (such as CREATE SCHEMA or CREATE INPUT WINDOW) immediately following it. Doc-comments not immediately preceding a top-level statement are seen as errors by the visual editor with SAP Sybase Event Stream Processor Studio.

Regular multi-line comments do not get treated specially by the Studio and may be used anywhere in the CCL project.

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 recognized
by the Studio visual editor and associated with the immediately
following statement (in this case the CREATE SCHEMA statement).
*/
CREATE SCHEMA S1 ...
```

The CREATE SCHEMA statement provided here is incomplete; it is shown only to illustrate that the doc comment is associated with the immediately following CCL statement.

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

```
/***********************************************************
Do not modify anything beyond this point without authorization
***********************************************************/
```

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.

CHAPTER 4   **CCL 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

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

Using the ADAPTER START statement, adapters can be put into startup groups, where each group is started sequentially. This is ensures that certain adapters are started, and load their data, before others.

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.

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
 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;
```

ATTACH ADAPTER Statement

Attach an adapter to a stream or window.

Syntax

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

Parameters

name	Name of 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 SAP 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>';
```

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 specified, it can be derived from the query.
primary_key_clause	Set primary key. See *PRIMARY KEY Clause* 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 *Queries* 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 on the output of a window, but a state does not need be maintained.

A delta stream typically forwards the opcode it receives. However, for a filter, a delta stream modifies 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.

See the <xref href="san1311804134032.xml">Using SPLASH in Flex Operators</xref> topic for more details.

Restrictions

- A delta stream cannot use functions that cannot be repeated, such as **random()** or **now()**. When a delta stream produces a delete record, the computed column in the record gets recalculated, and as a result, will not match what was originally computed and inserted for the record. Any downstream computation using this column could lead to incorrect results. An update is internally treated as a delete followed by an insert in many contexts and hence an update would also lead to the same issue for delta streams using non-repeatable functions.
- 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 deletes if they exist, otherwise they should be silently ignored.
- 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
;
```

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|LOCAL] [STREAM|DELTA STREAM|WINDOW] name schema_clause
[PRIMARY KEY (column1 [,...])] [store_clause] [keep_clause]
[aging_clause]
BEGIN
[local-declare-block]
ON input1 { [statement1 [,...]] };
[EVERY interval { [statement1 [,...]] };]
[ON START TRANSACTION { [statement1 [,...]] };]
[ON END TRANSACTION { [statement1 [,...]] };]
END;
```

Components

`procedureName`	The name of the Flex operator being created.
IN `input1`	Inputs to the Flex operator are declared in the **IN** clause. The inputs can be streams, delta streams, windows, or outputs of another flex operator.
KEEP `keep_spec`	The **KEEP** clause modifies the retention policy of existing input elements that are either delta streams or windows.
OUT `output_element`	The output of the Flex operator is defined in this clause. A Flex stream can have only one output. It can be a stream, a delta stream or a window, either local or output.
`name`	The name of the output element must be included in the **OUT** clause. This name is used when running queries against the flex operator.
`schema_clause`	This clause is mandatory for all output elements: stream, delta stream, and window. See *SCHEMA Clause* for more details.

PRIMARY KEY `(column1 [,...])`	(Optional) This clause may be used when specifying either a delta stream or a window as the output element. See *PRIMARY KEY Clause* for more details.
`store_clause`	(Optional) This clause may only be used when specifying a window as the output element. See *STORE Clause* for more details.
`keep_clause`	(Optional) This clause may only be used when specifying a window as the output element. See *KEEP Clause* for more details.
`aging_clause`	(Optional) This clause may only be used when specifying a window as the output element. See *AGING Clause* for more details.
GUARANTEE DELIVERY	(Optional) Enables guaranteed delivery for this window. Streams and delta streams do not support guaranteed delivery.
`local-declare-block`	(Optional) The **DECLARE** block can define variables and functions of all types, including complex data types such as records, vectors, dictionaries and event caches.
ON `input1`	The **ON** 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 **ON** input clause.
EVERY `interval`	(Optional) The SPLASH statements specified in this block are executed every time the interval expires. The interval can be specified explicitly, or specified through an interval type parameter.
ON START TRANSACTION	(Optional) The SPLASH statements specified in this block are executed at the start of each transaction.
ON END TRANSACTION	(Optional) The SPLASH statements specified in this block are executed at the end of each transaction.
`statement1`	(Optional) SPLASH statement to specify the processing logic.

Note: The **EVERY** interval must be:

- At least sixteen milliseconds on a Windows system
- At least one millisecond on any other system

If you enter a shorter interval, the compiler changes it to the system's minimum at run-time.

Usage

The **CREATE FLEX** statement creates 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, you can modify its retention policy 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

- You can use a **KEEP** clause 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 generate records with update or upsert opcodes. To generate records with these opcodes, use a Flex window instead of a Flex delta stream.
- You can use a SPLASH output statement 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. Guaranteed delivery is enabled on the output window so that subscribers to the window are assured of receiving all the output.

```
CREATE FLEX ComputeAveragePrice
 IN NASDAQ_Trades
 OUT OUTPUT WINDOW AverageTradePrice SCHEMA (Symbol string,
 AveragePrice money(4) ) PRIMARY KEY(Symbol) STORE store1 GUARANTEE
```

```
DELIVERY
 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;
```

CREATE LIBRARY Statement

In order to use external C/C++ and Java functions in CCL expressions, you must first declare them in your CCL project using the **CREATE LIBRARY** statement.

Syntax

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

Components

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 Linux or Solaris paths, preface the directory with a slash. For Windows paths, preface the directory with a double backslash.
	For Java functions, the name of the class file without the .class suffix. You can specify it as a string parameter. If running within Studio, specify the location of the class files by setting your CLASSPATH variable. If running outside of Studio, set the Java-classpath option in the project configuration file.
funcName	The name of the declared function.
returnType, argType	Datatype of the return value of the function and an argument of the function, respectively.

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.

Scalar functions are supported. Aggregate functions are not supported.

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 ESP datatypes, namely boolean, integer, long, float, money(n), date, bigdatetime, binary, string, interval, and timestamp.

Java external library calls support only integer, long, float, and string ESP 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 LIBRARY MyCFunctions LANGUAGE C FROM '/opt/sybase/
MyFunctions.so' (
    integer MyFunc1 (integer, integer, float);
    string MyFunc2(string);
);
```

Create a Java Function

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

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];
```

Parameters

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 input (source) streams being updated; it does not have an effect on derived (local or output) streams. A value of true guarantees that every record acknowledged by the system is persisted at the expense 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 **maxfilesize**.
ckcount	The maximum number of records written before writing the intermediate metadata. Default is 10,000.

Components

storename	An identifier that can be referenced in the **STORE** 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;
```

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.

Components

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

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 Store1 PROPERTIES INDEXTYPE='hash',
INDEXSIZEHINT=16;
```

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 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 {(columnname type [,...])|
       INHERITS [FROM] schema_name [,...] [(columnname 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);
```

CREATE SPLITTER Statement

The Splitter construct is a multi-way filter that sends data to different target streams depending on the filter condition. It works similar to the ANSI 'case' statement.

Syntax

```
CREATE [[LOCAL]|OUTPUT]  SPLITTER name AS
{ WHEN condition THEN {target_streamname [, …]} } […]
[ ELSE {target_streamname[,..]} ]
SELECT { column_list | * }
FROM source_name [{[alias] [KeepClause]}|{[KeepClause][alias]}]
;
```

Components

condition	Any expression that results in a 0 or 1.
name	Any string specified to identify the splitter construct. Must be unique within a module or top level project.
target_streamname	Name of a stream or delta stream into which the filtered records are inserted. Must be unique within the module or top level project.
source_name	The source (stream, window, or delta stream) that provides input data on which the splitter logic is applied.
column_list	A set of expressions referring only to the columns in the source stream, constant expressions, constant literals, global variables and functions, or parameters.

Usage

The target stream or delta streams are implicitly defined by the compiler. The schema for the target streams are derived based on the column_list specification. All the targets are defined as either local or output depending on the visibility clause defined for the splitter. The default is local. Note that when the splitter has an output visibility, output adapters can be directly attached to the splitter targets, even though those targets are implicitly defined.

Each filter condition in a splitter can have one or more target streams defined. However, each target stream name can appear only once in the list. This allows the possibility to send an event down multiple paths in the graph as the example below shows.

Note: When a condition evaluates to true, the following conditions are neither considered nor evaluated.

The semantics of the splitter are that of a switch statement. Whenever the condition evaluates to true (non-zero value), the record as projected in the column_list is inserted into the corresponding target streams. If the source is a:

- Stream, the targets are also streams.
- Delta stream or window, the targets are delta streams.

If the source is a window or delta stream, the primary keys need to be copied as-is. The other columns can be changed.

Note: When the source is a window or a delta stream, the warning about unpredictable results being produced if one of the projections contains a non-deterministic expressions that applies for delta streams also applies for splitters.

Local DECLARE BLOCKS cannot be specified on SPLITTERS. However, functions, parameters, and variables in the global DECLARE BLOCK can be accessed in the condition or column expressions in the projection.

Examples
Create a Splitter

In the following example, if a trade event arrives where the Symbol is IBM or ORCL, then the event is directed to both ProcessHardWareStock and ProcessSoftwareStock streams. If a trade event arrives where the Symbol is either 'SAP' or 'MSFT', then it is directed to the ProcessSoftwareStock stream. All other trades are directed to the ProcessOtherStock stream.

```
CREATE SPLITTER Splitter1 AS
WHEN Trades.Symbol IN ('IBM', 'ORCL' ) THEN ProcessHardWareStock,
ProcessSoftwareStock
WHEN Trades.Symbol IN ('SAP', 'MSFT') THEN ProcessSoftwareStock
ELSE ProcessOtherStock
SELECT * FROM Trades;
```

Performance Considerations
A splitter is typically more efficient both in terms of CPU utilization and throughput when there is more than a two way split than an equivalent construct composed of two or more streams that implement a filter. Unlike other streams in ESP, a Splitter and all its target streams run in a single thread. This means that the Splitter thread is responsible for distributing data to its dependents.

The Splitter is more efficient than its equivalent multi-threaded logic for these reasons:

- The performance of a stream is inversely proportional to the amount of data that a source stream needs to distribute to its target. If a stream has two dependent streams, it needs to

distribute twice the amount of data it produces (that is, one copy for each target stream). Similarly, if a stream has five dependencies it needs to distribute five times the data it produces. For example, this is the case when three filter streams depend on one source, with each filter only producing a third of the input data as output. In the case of a splitter, the source needs to distribute the data only once to the splitter and this reduces the load on the source stream.

- The decrease in CPU utilization comes from the fact that you don't have three separate streams processing 100% of the input data to produce, for example, a third of the data as output. In the case of the splitter, the incoming data is analyzed only once and typically no more than 100% of the incoming data is distributed to the appropriate target streams when the filter condition is satisfied.

However, note that because the splitter is single threaded, its performance advantage degrades quickly when it needs to distribute the same data more than once. For example, there is more than one target stream for each filter condition or when the target streams themselves have many dependents.

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]
[ autogenerate_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	(Optional) Can be specified on an input stream. This clause filters the events before accepting them from the adapter or an outside publisher. In the expression, reference column values in the form stream.column, where stream is the name of the stream being created by this statement, and column is the name of the column being referenced.

autogenerate_clause	(Optional) This can be used to automatically add a sequence number to each event. One or more columns are specified (datatype long) and the value in the column is incremented for each incoming event. Valid only for input streams. See *AUTOGENERATE Clause* for more information.
local-declare-block	Allows variable and function declarations that can be accessed in expressions in the query. You cannot define a local-declare-block on an input stream.
as_clause	For derived streams, this contains the continuous query (SELECT clause, FROM clause) that will define the output of this stream.

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 (Col1 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.Col1, InStr.Col2
FROM InStr
WHERE InStr.Col1 > 1000;
```

The following statement creates an input stream with auto generated values beginning at 100000 for the TradeId column, filtering out trades with prices below 1000. Note that the filtering is done after the TradeId is generated.

```
CREATE INPUT STREAM BigTrades
SCHEMA (TradeId long, Symbol string, Shares integer, Price money(4))
WHERE BigTrades.Price > 1000
AUTOGENERATE (TradeId) FROM 1000000;
```

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]
[autogenerate_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
;
```

Components

name	A name for the window being created.
schema_clause	Required for input windows, but optional for local 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.
autogenerate_clause	(Optional) Specify that the server will automatically generate values for one or more columns of datatype long. This can be used to generate a primary key for events that lack a natural key. Valid only for input windows. See *AUTOGENERATE Clause* for more information.
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 declarations that can be accessed in expressions in the query. You cannot define a local-declare-block on an input stream.
GUARANTEE DELIVERY	(Optional) Enables guaranteed delivery for this window.
as_clause	Introduces a query to a statement.
outfile_clause	
dumpfile_clause	

Usage

The **SCHEMA** and **PRIMARY KEY** clauses are 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 include 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.

Use **GUARANTEE DELIVERY** to enable guaranteed delivery only for a window stored in a log store; guaranteed delivery is not available to windows stored in memory stores. Do not use **GUARANTEE DELIVERY** inside a module because you cannot directly attach adapters to elements in modules.

Examples

This example creates a local window containing only position records received in the last ten minutes. It also uses the AGES clause to flag records that have not updated in the last 5 seconds by setting the value in the AgeColumn.

```
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 received in the last ten minutes. Inclusion of the local declare block reports 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;
```

The following statement creates a window that maintains only the last 1000 rows while also
getting updates on the age of the rows. The TradeId value is automatically generated
beginning at 0.

```
CREATE INPUT WINDOW FreshTrades
SCHEMA (TradeId long, Symbol string, Shares integer, Price money(4),
Age integer)
PRIMARY KEY (TradeId)
KEEP 1000 ROWS
AGES EVERY 5 MINUTES SET Age 100 TIMES
AUTOGENERATE (TradeId);
```

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]
```

- The**typeof()** operator provides a convenient way to declare variables. An example of the typeof usage would be: if rec1 is an expression with type [int32 key1; string key2; | string data;] then the declaration typeof(rec1) rec2; is the same as the declaration [int32 key1; string key2; | string data;] rec2;

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
;
```

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

```
create flex flex104
        in ds2_104
        out output stream flexos104 schema s1_104
        begin
            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;
```

IMPORT Statement

Import libraries, parameters, variables, and schema, function, and module definitions from another 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 **IMPORT** statement.

Usage

Only the following CCL statements are valid in an imported 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 Schema1
//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;
```

LOAD MODULE Statement

The LOAD MODULE statement loads a previously created module into the project. The CREATE MODULE statement can either be in the current CCL file or in an imported CCL file (see IMPORT statement).

Syntax

```
LOAD MODULE modulename AS moduleIdentifier
    in-clause
    out-clause
```

```
    [parameters-clause]
    [stores-clause];
```

Components

moduleName	The name of the module that must match the name of the previously created module.
module identifier	The name used to identify this instance of the module: it must be unique within the parent scope.
in-clause	Binds the input streams or windows defined in the module to previously-created streams or windows in the parent scope.
out-clause	Exposes one or more output streams defined within the module to the parent scope using unique identifiers.
parameters-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

The LOAD MODULE statement is used to create an instance of a previously defined module in the current project. The IN, OUT and optional PARAMETERS and STORES clauses bind the module to elements in the calling project. The same module may be "loaded" multiple times in a single project.

Before a module can be loaded, it must be defined in a CREATE MODULE statement in either the same project or an imported CCL file.

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 defines a module that processes raw stock trade information and outputs a list of trades with a price exceeding 1.00. The project then creates an instance of the module using the **LOAD MODULE** statement. The LOAD MODULE statement binds the project input stream "NYSEData" to the input stream of the module (TradeData) and creates a local stream called "NYSEPriceOver1Data" that is bound to the output stream of the module (FilteredTradeData).

```
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;
```

CHAPTER 5   **CCL 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 interval,specified in hours, minutes, seconds, milliseconds, or microseconds, after which the data aging process begins. It may be specified using a combination of the allowed 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
;
```

AS Clause

Introduces a CCL query to a derived element.

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

Components

AS	The **AS** 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 ( col1 string )
    AS
        SELECT inputStream.col1
        FROM inputStream;
```

AUTOGENERATE Clause

Specify one or more columns that will contain an automatically generated sequence number in records sent to an input stream or input window.

Syntax

```
AUTOGENERATE (column[, ...])[FROM {long_const|parameter}]
```

Components

column	Specify the name of the column in which the automatically generated value should be placed. The column must be of datatype long.
FROM	Overrides the default starting value of zero with the specified numeric constant or value found in the specified parameter at run time.

Usage

This clause can be used when specifying an input stream or input window in a project. It cannot be used when specifying an input stream or input window in a module. When input

records do not have a natural primary key, this clause provides a way to specify a column that can be used as the primary key.

You can specify more than one column that will have its value automatically generated. The column names must be unique. At run time all of the specified columns will get the same automatically generated value.

The automatically generated columns must be of type long. When the value exceeds the maximum positive value that the long datatype can hold, the value restarts from the maximum negative value that a long datatype can hold.

By default, the values start at zero and increase by one for each insert record. This can be overridden using the FROM clause to explicitly set a starting value, specified as either a parameter or a long_const.

An input window with an auto generated column may be assigned to a log store; in which case, on a restart, the next insert will get the highest sequence number recovered from the log store plus one as the value in the automatically generated column. When there is data in the log store, the FROM clause is ignored on restart.

The automatically generated column is only incremented on an insert and any value explicitly provided in the automatically generated columns of the input row on an insert is ignored. On an update, delete or upsert, the value in the auto generated column is used as it is provided in the input row. This rule has the potential to produce duplicate rows in a window. For example,

- The primary key is an auto generated column.
- On the first insert, the primary key is set to 0 because the key column is auto generated and the sequence number starts at 0.
- If the next row is an upsert with the primary key set to 1, the server will insert this row into the window because there is no row with the primary key of 1 to update.
- When another insert comes in, the server will set the auto incremented key column value to 1 and try to insert the row into the input window.
- This will cause a duplicate row in the store of the input window and the server will reject the record.

Therefore, it is recommended that the AUTOGENERATE clause not be used with an upsert opcode, especially when the automatically generated value is a primary key.

Examples
The following code creates an input stream named Trades with a column named TradeId for which the values are automatically generated.

```
CREATE INPUT STREAM Trades
SCHEMA (TradeId long, Symbol string, Shares integer, Price money(4))
AUTOGENERATE (TradeId);
```

This example creates an input window named Trades with a primary key column named TradeId for which the values are automatically generated.

```
CREATE INPUT WINDOW Trades
SCHEMA (TradeId long, Symbol string, Shares integer, Price money(4))
```

```
PRIMARY KEY (TradeId)
AUTOGENERATE (TradeId);
```

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 indicates 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
```

FROM Clause

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

FROM Clause: ANSI Syntax

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

Syntax

```
FROM { source [ [(DYNAMIC|STATIC)] [AS] alias] [keep_clause] |
nested_join }
[INNER|RIGHT|LEFT|FULL] JOIN
{ source [ [(DYNAMIC|STATIC)] [AS] alias] [keep_clause] |
nested_join }
on_clause
```

Components

source	The name of a data stream, window, reference, or delta stream.
DYNAMIC | STATIC	DYNAMIC indicates the data in the window or stream being joined is going to change often. A secondary index is created on windows joining with an incomplete primary key of a DYNAMIC window or stream. This improves performance but uses additional memory proportional to the total data length of key columns in the index. By default, windows and streams are STATIC and no secondary indices are created.
alias	An alternate name for the source.
keep_clause	An optional policy that specifies how rows are maintained in the window (it cannot be used with a reference, stream, or delta stream).
nested_join	A nested join — see below.

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 **FROM** to create inner, left , right, and full joins:

JOIN	If no join type is specified, the default is INNER.

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

The datasources used with this syntax can include data stream expressions, named and unnamed window expressions, queries, and references. You can use aliases for the 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

- Other clauses in the query can only refer to one of the datasources named in this clause.
- For a left outer join, the data stream must be on the left side.
- For a right outer join, the data stream must be on the right side.
- A full outer join cannot join a window to a data stream.
- A full outer join cannot include a reference.
- Only one of the datasources in a join can be a reference.
- A reference can't be on the outer side of the join.
- Joining a window to a reference without a primary key can only produce a stream or a window with aggregation.

FROM Clause: Comma-Separated Syntax

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

Syntax

```
FROM [ source [ [AS] alias ] [ keep_clause ] ] [, …]
```

Components

source	The name of a data stream, window, reference, or delta stream.
alias	(Optional) an alternate name for the source.
keep_clause	(Optional) A policy that specifies how rows are maintained in the window (it cannot be used with a reference, stream or delta stream).

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.

Restrictions

- Other clauses in the query can only refer to one of the datasources named in this clause.
- One reference cannot be directly joined to another (but, one reference can be joined to another datasource such as window, and the result of that join can be joined to another reference).
- Joining a window to a reference without a primary key can only produce a stream or a window with aggregation.

- When producing a window from the join, if the reference has a primary key, that entire key must be bound in the **WHERE** clause. And the resultant window cannot contain any columns from the reference in its primary key.

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 columns from the input window or stream. However, an expression cannot use aggregate functions.

Usage

It combines one or more input rows into a single row of output. Typically, **GROUP BY** is used for aggregation. The query will contain a **GROUP BY** to specify how to group the inputs, and one or more of the column expressions in the **SELECT** clause will use an aggregate function to compute aggregate values for the group.

When a **GROUP BY** clause is used in a query, the compiler will deduce the primary key based on the group by expression(s). 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.

Note that the **GROUP BY** clause must reference input columns directly. It cannot use aliases defined in the local **SELECT** clause.

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 BY T.Symbol
```

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 aggregate functions such as **min()** or **max()**. The expression 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';
```

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';
```

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';
```

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;
```

KEEP Clause

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

Syntax

```
KEEP {{[EVERY] count ROW[S][SLACK slackcount][PER(col1[,…])]} | ALL
[ROW[S]]}
```

```
KEEP [EVERY] interval [PER (col1[,…])]
```

```
KEEP [EVERY] { 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 a parameter or a constant integer.
	Note: The SLACK component cannot be used with the EVERY modifier.
time_policy	Specify the length of time that records will be retained in the window as described in the *Intervals* on page 24 topic.
ALL	Specifies that all of the rows received will be retained.
EVERY	Specifies that when the maximum number of records is exceeded or the time interval expires, every retained record is deleted. When this modifier is used, the resulting window is a Jumping Window. Otherwise, the resulting window is a Sliding Window
PER	Specifies that the retention policy will be applied to groups of rows rather than at the window level.

Note: When specifying **EVERY** interval, the interval must be at least one millisecond. If you enter a shorter interval, the compiler will change it to one millisecond. Additionally, on Windows systems the interval must be at least sixteen milliseconds. If you enter a shorter interval it will be changed to sixteen milliseconds at run-time.

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.

Including the **EVERY** modifier in the **KEEP** clause produces a Jumping Window, which deletes all of the retained rows when the time interval expires or a row arrives that would exceed the maximum number of rows.

Specifying the **KEEP** clause without the **EVERY** modifier produces a Sliding Window, which deletes individual rows once a maximum age is reached or the maximum number of rows are retained. Specifying a **SLACK** value causes the retention mechanism to get triggered when the number of stored rows equals (count+slackcount) as opposed to count. When specifying a Sliding Window with 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 is the case where there is a **KEEP** clause attached to the **FROM** clause of the query.

Note: The SLACK value cannot be used with the EVERY modifier, and thus cannot be used in a Jumping Windows retention policy.

Use the **PER** sub-clause within the **KEEP** clause syntax to retain data based on content for both named and unnamed windows. The feature supports both row-based and time-based retention. Rather than applying the retention policy at the window level, it will be applied to individual groups of rows based on the PER expression.

Note that unnamed windows can be created on delta streams or windows, but they cannot be created on streams. Windows on streams must be created explicitly using a CREATE WINDOW statement.

The following example creates a sliding window that retains 2 rows for each unique value of Symbol. Once 2 records have been stored for any unique Symbol value, arrival of a third record (with the same Symbol value) will result in deletion of the oldest stored record with the same Symbol value.

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

CREATE INPUT WINDOW TradesWin1
    SCHEMA TradesSchema
    PRIMARY KEY(Id)
    KEEP 2 ROWS PER(Symbol)
;
```

The following example creates a jumping window that retains 5 seconds worth of data for each unique value of Symbol.

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

CREATE INPUT WINDOW TradesWin2
    SCHEMA TradesSchema
    PRIMARY KEY(Id)
```

```
    KEEP EVERY 5 SECONDS PER(Symbol)
;
```

MATCHING Clause

This is used within a derived stream's 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
```
MATCHING [interval:pattern]
ON { {source.column = source.column [=...]}|
                {source.column = constant }|
                {getOpcode() = opcode_constant} [AND...]
          }

pattern:[!]{event | (event)} [{&&| || |,}event]
```

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. The options are: INSERT, UPDATE, DELETE.
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 elements that are used as inputs for pattern matching.

Note: If a pattern requires multiple events from the same input element, it may be preferable to use aliases to create a unique reference for each event.

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. An event is a row arriving from a specified input element. 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 conditions 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 members. 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, with the components on the left of the operator occuring before the components on the right, 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 getOpcode ( ) function .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 getOpcode ( ) function, the right hand side must contain a constant specifying the desired opcode. Valid opcode values are insert, update and delete.

For details on pattern matching, see the *SAP Sybase Event Stream Processor Programmers Guide*

ON Clause: Join Syntax

Specifies the conditions that must be met to perform a join of the data from two sources.

Syntax

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

Components

source	The names of the sources in the **FROM** 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, delta streams, or references) 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 datasources of the join. The comparison cannot specify a literal value, or compare two columns in the same data source.
- For a left outer join, the data stream must be on the left side.
- For a right outer join, the data stream must be on the right side.

- Other clauses in the query can only refer to one of the datasources named in this clause.
- One reference cannot directly join another (but, one reference can join another datasource such as a window, and the result of that join can join another reference).
- A reference can't be on the outer side of the join.
- A full outer join cannot include a reference.
- A full outer join cannot join a window to a data stream.
- Joining a window to a reference without a primary key can only produce a stream or a window with aggregation.

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 `output1-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;
```

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 := 4;
        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;
```

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.

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 need to be 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

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 **CREATE SCHEMA** 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**.

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.

STORE Clause

Assigns the store for the window in any window definition.

Syntax
```
STORE storename
```

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.

store1-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;
```

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 **SELECT** 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, a1 STRING, a2 string);
CREATE SCHEMA MySchema2 (a0 integer, a1 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
```

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')
;
```

CHAPTER 6   **CCL Functions**

A function is a self-contained, reusable block of code that performs a specific task.

The SAP 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.

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.

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 )
```

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

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.

atan2()

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 )
```

Parameters

expression1	Expression that simplifies to an integer or a long (must be the same datatype as **expression2**).
expression2	Expression that simplifies to an integer or a long (must be the same datatype as **expression1**).

Usage
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 )`

Parameters

bit	An integer indicating which bit to set, starting from 0 as the least-significant bit.

Usage

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

`bitflaglong (35)` returns 34359738368 or
100000000000000000000000000000000000 in binary.

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

`bitmasklong (33, 35)` returns 60129542144 or
11100000000000000000000000000000000000 in binary.

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
11111111111111111111111111111000. The user cannot specify binary directly.

bitor()

Scalar. Returns the results of performing a bitwise OR operation on two expressions.

Syntax

```
bitor ( expression1, expression2 )
```

Parameters

expression1	Expression that simplifies to an integer or a long (must be the same as **expression2**).
expression2	Expression that simplifies to an integer or a long (must be the same as **expression1**).

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.

Usage
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 )
```

Parameters

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 )
```

Parameters

expression1	Expression that simplifies to an integer or a long (must be the same datatype as **expression2**).
expression2	Expression that simplifies to an integer or a long (must be the same datatype as **expression1**).

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 )
```

Parameters

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.

concat()

Returns the concatenation of two given binary values OR one or more string values.

Syntax

```
concat ( binary1, binary2 )
concat ( string1, ...stringn)
```

Parameters

binary1	A binary value.
binary2	A binary value.
string1	The first string value in the set.
stringn	The final string value in the set.

Usage

When working with binaries, concatenates the given binary arguments into a single binary and returns that value. The function returns NULL if either argument is NULL.

When working with strings, concatenates the given string arguments into a single string and returns that value. Literal text must be enclosed in single quotation marks.

Example

```
concat ( hex_binary ('aabbcc'), hex_binary ('ddeeff') )
```
returns AABBCCDDEEFF.

```
concat ( hex_binary ('ddeeff'), hex_binary ('aabbcc') )
```
returns DDEEFFAABBCC.

```
concat ( 'MSFT', '_NYSE')
```
returns MSFT_NYSE.

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 ( point1x, point1y, [point1z], point2x, point2y,
[point2z] )
```

Parameters

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.

Usage

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 ( point1x, point1y, [point1z], point2x, point2y,
[point2z] )
```

Parameters

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.

Usage

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.

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.

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 )
```

Parameters

expression	An expression of any datatype.

Usage
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
```
isnull ('examplestring') returns 0.
```

ln()
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	An expression that evaluates to a float greater than or equal to 0.

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 [,...] )
```

Parameters

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.

pi()

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 **base** 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 )
```

Parameters

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.

Usage

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.

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.

sinh()

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'.

length()

Scalar. Returns the number of bytes of a given binary value, string, or multibyte string.

Syntax
```
length ( expression )
```

Parameters

expression	A binary value, string, or multibyte string.

Usage

Returns the number of bytes that make up a given value. The function can take a binary value, string, or multibyte string as its argument, and it returns an integer. If the argument is NULL, the function returns NULL.

Note: To make the function return the number of characters in multibyte strings, enable the utf8 project configuration option. For more information on how to configure the utf8 option, see the *Studio Users Guide* and the *Configuration and Administration Guide*.

Example
```
length (hex_binary ('0xaa1234')) returns 3.
```
```
length (hex_binary ('aa')) returns 1.
```
```
length ('ABCDE') returns 5.
```
```
length ('字节') returns 6 (with utf8 disabled) or 2 (with utf8 enabled).
```

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('longlonglongstring', '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('Stringi', 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 optional 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('aaaaabaaaabbbaaa', '[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 )
```

Parameters

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.

Usage
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'.
```

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.

Usage
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 )
```

Parameters

character	A character string.

Usage

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 base64-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.

char()
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'.

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.

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 ('a1b2c3e4'),1,2) returns B2C3.

extract ( hex_binary ('a1b2c3e4'),3,1) returns E4.

extract ( hex_binary ('a1b2c3e4'),0,4) returns A1B2C3E4.
```

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 (FFFFFFF6) 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.
```

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.
```

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.

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.

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 (midnight, 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.

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'.

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 bigdatetime to the number of seconds since the epoch (midnight, January 1, 1970).

Syntax

```
timeToSec ( time )
```

Parameters

time	A bigdatetime.

Usage

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_bigdatetime()

Scalar. Converts a given value to a bigdatetime.

Syntax

```
to_bigdatetime ( value )
to_bigdatetime ( value, format )
```

Parameters

value	A string, float, long, or bigdatetime. Strings must be in the format specified by the format argument. The default format is yyyy-mm-dd hh:mm:ss.xxxxxx.
	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.

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(3600000000)` 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.

See also

- *Date/Time Format Codes* on page 252

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_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 seconds 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 )
```

Parameters

value	A string, interval, date/time type, numeric type, or money type.

Usage
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 seconds, milliseconds, or microseconds from the epoch (midnight, January 1, 1970 UTC) depending on whether the input type is a date, timestamp or bigdatetime respectively. 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 seconds, milliseconds, or microseconds from the epoch (midnight, January 1, 1970 UTC) depending on whether the input type is a date, timestamp or bigdatetime respectively. 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_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 milliseconds 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.

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.

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.

tonetbinary()

Scalar. Converts an integer in host byte order to a 4 byte binary in network byte order.

Syntax

```
tonetbinary ( integer )
```

Parameters

integer	An integer in host byte order.

Usage

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.

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

```
xmlelement (top, xmlattributes('data' as attr1),
```
xmlparse('<t/>')) returns a new XML element called top, with a 'data' attribute and
child element.

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 )`

Parameters

calendar	A string representing the file path for a calendar file
datevalue	A date/time type

Usage

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] )
```

Parameters

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.

Valid Date Part Keywords and Multiples

Keyword	Keyword meaning	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.

Usage

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
SAP 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] )
```

Parameters

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.

Valid Date Part Keywords and Multiples

Keyword	Keyword meaning	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.

Usage

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

SAP 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'
```

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.
```

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:
	• The year, if the string is yy or yyyy. • The month, if the string is mm or m. • The day of the year, if the string is dy or y. • The day of the month, if the string is dd or d. • The day of the week, if the string is dw. • The hour, if the string is hh. • The minute, if the string is mi or n. • The second, if the string is ss or s.
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] )
```

Parameters

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.

Valid Date Part Keywords and Multiples

Keyword	Keyword meaning	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.

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 >= 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.
```

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 negative 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.
```

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.

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.

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.

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.

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.

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.
```

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;
```

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 expression 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:

$$\overline{X} = \frac{\sum X}{N}$$

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 )
```

Parameters

dependent-expression	The variable that is affected by the independent variable. 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.
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 **expression** typically based upon columns from the source. It can also be a **constant expression**.

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 **α**, a number between 0 and 1. **α** may be expressed as a percentage, so a smoothing factor of 10% is equivalent to **α**=0.1. Alternatively, **α** may be expressed in terms of **N** time periods. For example,

$$\alpha = \frac{2}{N+1}$$

N=19 is equivalent to α=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 α; smaller α values make the choice of $S2$ relatively more important than larger α values, since a higher α 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 )
```

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

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 )
```

Parameters

numeric-expression	Expressions include integer, long, float, money, timestamp, and interval types.

Usage

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.

$$WMA_M = \frac{np_M + (n-1)p_{M-1} + \cdots + 2p_{M-n+2} + p_{M-n+1}}{n + (n-1) + \cdots + 2 + 1}$$

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 expression 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 tendency 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 tendency). 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 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. 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 )
```

Parameters

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.

When assessing stock portfolios, use the **nth** function to identify a specific item in a list. For example, you can identify the day's third-highest traded stock price indicated by the third item in the index. The nth function's uses a 0-based index.

Note: If the **number** argument is greater than the number of elements in the group, this function returns a NULL value.

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 )
```

Parameters

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 *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 )
```

Parameters

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 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 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 timestamp, 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 timestamp, 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 )
```

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 timestamp, bigdatetime, and interval.

Parameters

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 timestamp, 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 )
```

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 timestamp, 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 )
```

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 timestamp, 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 ) * 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 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 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:

$$s = \sqrt{\frac{\sum (\bar{x} - x_i)^2}{n-1}}$$

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 )
```

Parameters

expression	The object that is summed. The expression accepts all datatypes except boolean.

Usage
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 from a group, based on the last row applied into that group. Returned values include NULLS.

Syntax

```
valueinserted ( expression )
```

Parameters

expression	The expression accepts all datatypes.

Usage

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.

Example

The following is an aggragate that outputs the minimum, maximum, average, and last value of the rows read from `inWin`, grouped by symbol. The last value is called by using the valueinserted() function.

```
 CREATE INPUT WINDOW inWin SCHEMA  ( id long, symbol string, value
money(2) ) PRIMARY KEY ( id ) KEEP 5 MINUTES;

 CREATE OUTPUT WINDOW stats PRIMARY KEY DEDUCED KEEP 5 MINUTES  AS
 SELECT i.symbol,
        min(i.value) minValue,
        max(i.value) maxValue,
        avg(i.value) avgValue,
        valueInserted(i.value) lastValue
 FROM inWin i GROUP BY i.symbol;
```

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. **expression** 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:

$$s^2 = \frac{\sum\left(X_i - \overline{X}\right)^2}{n}$$

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. **expression** 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 variance (*s2*) of numeric-expression (*x*) is computed according to this formula, which assumes a normal distribution:

$$s^2 = \frac{\sum\left(X_i - \overline{X}\right)^2}{n}$$

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:

$$Pvwap = \frac{\Sigma j \; Pj \cdot Qj}{\Sigma j Qj}$$

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:

$$WMA_M = \frac{np_M + (n-1)p_{M-1} + \cdots + 2p_{M-n+2} + p_{M-n+1}}{n + (n-1) + \cdots + 2 + 1}$$

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) )
```

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 a long.

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(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(qTrades.Price,2) ) {
```

```
                  deleteCache(tradesCache, counter);
                  insertCache( tradesCache, qTrades );
                  return rec;
                  break;
               } else if( qTrades.Price < rec.Price) {
                  break;
               }
               counter++;
         }
         if(cacheSz < 3) {
            insertCache(tradesCache, qTrades);
            return qTrades;
         } else {
            rec := getCache(tradesCache, 0);
            deleteCache(tradesCache, 0);
            insertCache(tradesCache, qTrades);
            return rec;
         }
         return null;
      }
   END;

   ON QTrades {
      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;
```

coalesce()

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'.

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 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(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(qTrades.Price,2) ) {
                    deleteCache(tradesCache, counter);
                    insertCache( tradesCache, qTrades );
                    return rec;
                    break;
                } else if( qTrades.Price < rec.Price) {
                    break;
                }
                counter++;
            }
            if(cacheSz < 3) {
                insertCache(tradesCache, qTrades);
                return qTrades;
            } else {
                rec := getCache(tradesCache, 0);
                deleteCache(tradesCache, 0);
                insertCache(tradesCache, qTrades);
                return rec;
            }
            return null;
        }
    END;

    ON QTrades {
        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;
```

firstnonnull()

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 )
getintegercolumnbyindex ( record, colname )
getdatecolumnbyindex ( record, colname )
gettimestampcolumnbyindex ( record, colname )
getbigdatetimecolumnbyindex ( record, colname )
getintervalcolumnbyindex ( 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 )
getfloatcolumnbyname ( 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 **colname** argument for **getstringcolumnbyname()** 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(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(qTrades.Price,2) ) {
                        deleteCache(tradesCache, counter);
                        insertCache( tradesCache, qTrades );
                        return rec;
                        break;
                    } else if( qTrades.Price < rec.Price) {
```

```
                  break;
              }
              counter++;
          }
          if(cacheSz < 3) {
              insertCache(tradesCache, qTrades);
              return qTrades;
          } else {
              rec := getCache(tradesCache, 0);
              deleteCache(tradesCache, 0);
              insertCache(tradesCache, qTrades);
              return rec;
          }
          return null;
      }
    END;

    ON QTrades {
        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()

Provides a way of interacting with databases: extracts data, calls stored procedures, and inserts data.

Syntax

```
getData(vector, service, query[, expr[, ...]])
```

Parameters

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)

Usage

Specify the name of the vector you want the records returned in 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 *Configuration and Administration Guide* for more information about this file and the services described in it.

Specify the query parameter 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. How you want to use the getData () function affects what you specify. The following table provides instruction for each use.

When	In the query parameter.
Extracting Data	Specify the database query to make callouts to databases. 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.
Calling Stored Procedures	Specify the appropriate command to execute the stored procedure. Provide any input parameters to the procedure as additional arguments.
	Note: If the stored procedure returns a result set, its schema needs to match that of the vector parameter to store the results. If the stored procedure only performs insertion in the tables, than the dummy row datatype must match the return value.
Inserting Data	Issue an INSERT statement to insert data into databases. Provide any parameters as additional arguments.

Example
Project logic looks for certain conditions and invokes getData() only when those conditions are met.

```
getData(v, 'MyService', 'SELECT col1, 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 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 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()

Returns the sequence number, or ROWID, of a given row in the window or stream as a 64-bit integer.

Syntax

```
getrowid ( row )
```

Parameters

row	A record from a window, stream, or delta stream, which is identified by the window or stream's name.

Usage

Each incoming row of data is assigned a unique sequence number or ROWID. The getrowid function takes a window or stream's identifier as its argument, and returns the ROWID of the row in the window or stream. It can be used with a stream, delta stream, or window. For more information about ROWID, see the *Programmers Guide*.

Example

```
CREATE INPUT STREAM MarketIn
     SCHEMA ( Symbol STRING, Price MONEY(4), Volume INTEGER );

CREATE  OUTPUT STREAM MarketOut
     SCHEMA ( RowId LONG, Symbol STRING, Price MONEY(4), Volume
INTEGER, TradeValue MONEY(4))
AS
     SELECT
         getrowid ( MarketIn ) RowId,
         MarketIn.Symbol,
         MarketIn.Price,
         MarketIn.Volume,
         MarketIn.Price * MarketIn.Volume as TradeValue
     FROM MarketIn;
```

rank()

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 FILTER** expression. This function has no arguments, and the function returns an integer.

Example

`rank() > 3` returns 0 for the first four rows in a group and 1 for all other rows.

sequence()

Combines two or more expressions to be evaluated in order.

Syntax

```
sequence ( expression [, ...] )
```

Parameters

expression	An expression of any data type. The last expression in the sequence determines the type and value for the entire sequence.

Usage

Combines two or more expressions to be evaluated in order. The type and value of the expression is the type and value of the last expression.

Sequencing is useful in a projection list to perform several simple instructions in the context of evaluating a projection column value without having to write a SPLASH UDF.

Example

This example computes the maximum price seen so far, assigns it to the `maxPrice` variable, and returns the product of the maximum price and number of shares.

```
sequence (
        maxPrice := case when maxPrice <
inRec.Price then inRec.Price else maxPrice end;
maxPrice*inRec.Shares
)
```

User-Defined External Functions

In CCL projects, use the **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 Solaris, 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.

Scalar functions are supported. Aggregate functions are not supported.

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

Java external library calls only support integer, long, float, and string ESP 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 SAP 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    int32v;
        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.

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);
   }
   ```

   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: Use the **setX** function to set the return value, where X is the return type of the return value. Using the **setX** function ensures that the null flag is set to false. To set the return value to NULL, say **top->null = true**.

The extern declaration ensures the function has the same name within the library and not the C++ function name.

The __DLLEXPORT__ preprocessor macro must be defined under Windows to make the external function available to ESP.

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, float, float);
);
```

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.

4. Call the distance function in the project using DistanceLib.distance(arg1, arg2, arg3).

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 1.7 runtime environment is included with SAP 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 and Solaris, and jvm.dll on Windows.

For example, to set the variable on a Linux 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(float, float,
    float) {
        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 class file:

```
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' (
    float distance(float, float, float);
);
```

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.

4. Call the function in the project using `DistanceLib.distance(float, float, float)`.

5. Link the Java library to the Event Stream Processor Server.

   If running within Studio, specify the location of the class files by setting your CLASSPATH variable. If running outside of Studio, set the Java-classpath option in the project configuration file.

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.

CHAPTER 7   **Programmatically Reading and Writing CCL Files**

Using the CCL read/write SDK, you can create new CCL files, read existing files, and modify the CCL statements within files with a set of SDK calls.

You can open, read, and write CCL files using a set of Java classes that allows you to manipulate a CCL parse tree programmatically. You can create custom tools that interact with CCL files (such as a translator from CCL to a different file format or a user interface to visualize CCL files) without also having to create your own parser and pretty-printer to manipulate CCL code as they have already been built in the SDK.

The CCL read/write SDK is constructed using the same Eclipse technologies (XTEXT and EMF) that Studio visual and text editors use to manipulate CCL files. The programs and examples created within this SDK can be run in a standalone manner outside of the Eclipse IDE.

CCL File Creation

The example below performs the necessary initialization and demonstrates how to create a new CCL file named `hello.ccl` with a single `CREATE INPUT STREAM` CCL statement. Note that all the Java code is necessary for file creation except for the three lines involving the Input Stream statement.

```
package com.sybase.esp.ccl.example1;

import java.io.File;

import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.ecore.resource.Resource;
import org.eclipse.xtext.resource.SaveOptions;
import org.eclipse.xtext.resource.XtextResourceSet;

import com.sybase.esp.CclStandaloneSetup;
import com.sybase.esp.ccl.CclFactory;
import com.sybase.esp.ccl.CclPackage;
import com.sybase.esp.ccl.InputStream;
import com.sybase.esp.ccl.Statements;

public class HelloCcl {

    public static void main(String[] args) {
        // This call must be made once in order to use the CCL API.
        CclStandaloneSetup.doSetup();
```

```
        // The file to be created. If it exists, remove it.
        String theFile = "hello.ccl";
        File cclFile = new File(theFile);
        if(cclFile.exists())
        {
            cclFile.delete();
        }

        // Ccl elements need to be placed in a Resource which is
        // within a ResourceSet. This is default EMF behavior.
        XtextResourceSet myResourceSet = new XtextResourceSet();
        URI uri = URI.createFileURI(theFile);
        Resource resource = myResourceSet.createResource(uri);

        // Use the CclFactory to create new Ccl elements, this is a
        // standard way EMF creates new elements in the Ccl API.
        CclFactory fact = CclPackage.eINSTANCE.getCclFactory();

        // Statements is the root object for the Ccl model.
        // Create one and add it to the Resource.
        Statements root = fact.createStatements();
        resource.getContents().add(root);

        // Create and name the InputStream.
        InputStream theInput  = fact.createInputStream();
        theInput.setName("NewInput");

        // Add the InputStream to the Stmts collection.
        root.getStmts().add(theInput);


        // Save an EMF Resource named hello.ccl
        try
        {
            SaveOptions saveOptions =
SaveOptions.newBuilder().getOptions();
            resource.save(saveOptions.toOptionsMap());
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

The output file of the above example, hello.ccl, contains a single CCL statement and can be seen below.

```
CREATE INPUT STREAM NewInput ;
```

CCL File Deconstruction

The SDK contains several different resources and methods to read, analyze, and output the contents of a CCL file.

The **walkModel** method below opens a CCL file and deconstructs it by iterating through each CCL statement and printing information on any affected CCL elements. The method then calls the **prettyPrint** procedure to print the statements themselves to System.out in CCL plain text.

```
public void walkModel(String theFile)
{
    XtextResourceSet myResourceSet = new XtextResourceSet();
    URI uri = URI.createFileURI(theFile);
    Resource resource = myResourceSet.getResource(uri, true);
    EcoreUtil.resolveAll(resource);

    Statements root = (Statements)resource.getContents().get(0);
    List <TopStatement> stmnts = root.getStmts();
    for(TopStatement d: stmnts)
    {
        printCclName(d);
    }
    prettyPrint(root);
}

void prettyPrint(EObject theEO)
{
    try
    {
        ISerializer serializer = getSerializer();
        if(serializer==null)
        {
            System.out.println("Injection bug");
        }
        else
        {
            System.out.println(serializer.serialize(theEO));
        }
    }
    catch(Exception e)
    {
        System.out.println(e.getMessage());
    }
}
```

Below is a sample CCL file containing several statements.

```
DECLARE
    PARAMETER integer the_integer := 1; PARAMETER boolean
the_boolean := FALSE;
END;
CREATE SCHEMA NewSchema ( col_0 integer , col_1 integer , col_2
```

```
integer , col_3 integer , col_4 integer , col_5 integer , col_6
integer ,
col_7 integer , col_8 integer , col_9 integer );
CREATE SCHEMA NewSchema2 ( AAAAA integer );
CREATE INPUT STREAM NewInputStream SCHEMA NewSchema;
CREATE INPUT WINDOW NewInputWindowWithInlineSchema SCHEMA ( c_key
integer , c_1 integer , c_2 long , c_3 string ) PRIMARY KEY ( c_key );
CREATE INPUT WINDOW NewInputWindow SCHEMA NewSchema PRIMARY KEY
( col_0 ) KEEP ALL ROWS;
CREATE OUTPUT WINDOW NewDerivedWindow PRIMARY KEY DEDUCED AS SELECT *
FROM NewInputWindow IN1;
CREATE OUTPUT STREAM NewDerivedStream AS SELECT * FROM NewInputStream
IN1;
CREATE FLEX NewFlex IN NewInputStream OUT OUTPUT WINDOW NewFlex
SCHEMA NewSchema PRIMARY KEY ( col_0 )
BEGIN
    ON NewInputStream {
    };
END;
CREATE OUTPUT SPLITTER NewSplitter AS WHEN 1 THEN NewSplitter_Output
SELECT * FROM NewInputWindow;
CREATE INPUT WINDOW JoinInputWindow1 SCHEMA NewSchema PRIMARY KEY (
col_0 ) KEEP ALL ROWS;
CREATE INPUT WINDOW JoinInputWindow2 SCHEMA NewSchema2 PRIMARY KEY (
AAAAA ) KEEP ALL ROWS;
CREATE OUTPUT WINDOW NewJoinWindow PRIMARY KEY ( AAAAA ) AS SELECT *
FROM JoinInputWindow1 J1 INNER JOIN JoinInputWindow2 J2 ON J1.col_0 =
J2.AAAAA;
CREATE INPUT STREAM NewInputStream1 SCHEMA NewSchema2;
CREATE INPUT STREAM NewInputStream2 SCHEMA NewSchema2;
CREATE OUTPUT STREAM NewUnionStream AS SELECT * FROM NewInputStream1
U1 UNION SELECT * FROM NewInputStream2 U2;
CREATE OUTPUT ERROR STREAM NewErrorStream ON NewUnionStream;
CREATE OUTPUT STREAM NewDerivedStreamSelective AS SELECT IN1.col_0 ,
IN1.col_1 , IN1.col_2 , IN1.col_3 , IN1.col_4 , IN1.col_5 , IN1.col_6
, IN1.col_7 , IN1.col_8 , IN1.col_9 FROM NewInputStream IN1;
CREATE OUTPUT STREAM NewDeriveStreamWithPattern AS SELECT * FROM
NewInputStream IN1 MATCHING [ 1 SECOND : IN1 ];
CREATE OUTPUT WINDOW NewCommaJoinWindowWithInputs PRIMARY KEY
DEDUCED AS SELECT * FROM JoinInputWindow1 input_1 , JoinInputWindow2
input_2;
```

After calling the **walkModel** method with the above CCL file as the argument, the information outputted by the **printCclName** procedure is as follows:

```
NewSchema kind = Schema
    col_0    integer
    col_1    integer
    col_2    integer
    col_3    integer
    col_4    integer
    col_5    integer
    col_6    integer
    col_7    integer
    col_8    integer
    col_9    integer
```

```
NewSchema2 kind = Schema
    AAAAA    integer
NewInputStream kind = InputStream
NewInputWindowWithInlineSchema kind = InputWindow
NewInputWindow kind = InputWindow
NewDerivedWindow kind = Window
NewDerivedStream kind = Stream
NewFlex kind = FlexOperator
NewSplitter kind = Splitter
JoinInputWindow1 kind = InputWindow
JoinInputWindow2 kind = InputWindow
NewJoinWindow kind = Window
NewInputStream1 kind = InputStream
NewInputStream2 kind = InputStream
NewUnionStream kind = Stream
NewErrorStream kind = ErrorStream
NewDerivedStreamSelective kind = Stream
NewDeriveStreamWithPattern kind = Stream
NewCommaJoinWindowWithInputs kind = Window
NewDerivedWindowWithWhere kind = Window
```

CHAPTER 8   **SPLASH Programming Language**

This chapter describes the Streaming Platform LAnguage SHell (SPLASH), which is a scripting language supported by SAP ESP that brings extensibility to CCL. It is used to define custom functions, custom operators in the form of Flex Operators, and is used to declare global and local variables and data structures.

The syntax of SPLASH is a combination of the expression language and a C-like syntax for blocks of statements. Just as in C, there are variable declarations within blocks, and statements for making assignments to variables, conditionals and looping. Other datatypes, beyond scalar types, are also available within SPLASH, including types for records, collections of records, and iterators over those records. Comments can appear as blocks of text inside /*-*/ pairs, or as line comments with //.

Variable and Type Declarations

SPLASH variable declarations resemble those in C: the type precedes the variable names, and the declaration ends in a semicolon. The variable can be assigned an initial value as well.

Here are some examples of SPLASH declarations:

```
integer a, r;
float b := 9.9;
string c, d := 'dd';
[ integer key1; string key2; | string data; ] record;
```

The first three declarations are for scalar variables of types `integer`, `float`, and `string`. The first has two variables. In the second, the variable "b" is initialized to 9.9. In the third, the variable "c" is not initialized but "d" is. The fourth declaration is for a record with three columns. The key columns "key1" and "key2" are listed first before the | character; the remaining column "data" is a non-key column. The syntax for constructing new records is parallel to this syntax type.

The `typeof` operator provides a convenient way to declare variables. For instance, if rec1 is a record with type [integer key1; string key2; | string data;].

```
typeof(rec1) rec2;
```

The above declaration is the same as the following declaration:

```
[ integer key1; string key2; | string data; ] rec2;
```

SPLASH type declarations also resemble those in C. The `typedef` operator provides a way to define a synonym for a type expression.

```
typedef float newFloatType;
typedef [ integer key1; string key2; | string dataField; ] rec_t;
```

These declarations create new synonyms `newFloatType` and `rec_t` for the float type and the given record type, respectively. Those names can then be used in subsequent variable declarations which improves the readability and the size of the declarations:

```
newFloatType var1;
rec_t var2;
```

Custom Functions

You can write your own functions in SPLASH. They can be declared in global blocks, for use by any stream or window, or within a local block to restrict usage to the local stream/window. A function can internally call other functions, or call themselves recursively.

The syntax of SPLASH functions resembles C. In general, a function looks like:

```
type functionName(type1 arg1, ..., typen argn) { ... }
```

Each "function type" is a SPLASH type, and each `arg` is the name of an argument. Within the `{...}` can appear any SPLASH statements. The value returned by the function is the value returned by the `return` statement within.

Here are some examples:

```
integer factorial(integer x) {
   if (x <= 0) {
     return 1;
   } else {
     return factorial(x-1) * x;
   }
 }
string odd(integer x) {
   if (x = 1) {
      return 'odd';
   } else {
      return even(x-1);
   }
 }

string even(integer x) {
   if (x = 0) {
     return 'even';
   } else {
    return odd(x-1);
   }
}
integer sum(integer x, integer y) { return x+y; }
string getField([ integer k; | string data;] rec) { return rec.data;}
```

The first function is recursive. The second and third are mutually recursive; unlike C, you do not need a prototype of the "even" function in order to declare the "odd" function. The last two functions illustrate multiple arguments and record input.

The real use of SPLASH functions is to define, and debug, a computation once. Suppose, for instance, you have a way to compute the value of a bond based on its current price, its days to maturity, and forward projections of inflation. You might write this function and use it in many places within the project:

```
float bondValue(float currentPrice,
                integer daysToMature,
                float inflation)
{
  ...
}
```

Using SPLASH in Flex Operators

Procedures written in SPLASH are integrated into Projects using the CCL Flex operator.

Procedures written in SPLASH are not meant to be standalone programs. They are meant to be used in SAP Sybase Event Stream Processor projects that are primarily written in CCL. The Flex Operator is the CCL statement that incorporates a SPLASH routine into a CCL project.

Operations on Windows that are inputs to the Flex Operator

- **Get value by key** – Get a record from the window by key. If there is no such key in the window, return null.

  Syntax: `windowValue[ recordValue ]`

  Type: The recordValue must have the record type of the window. The operation returns a value of the record type of the window.

  Example: `input_window[ [k = 3; | ] ]`

  Note: Non-key fields of the argument do not matter. The operation returns a record with the current values of the non-key fields, if a record with the key fields exists.

  If a key field is missing from the argument, or the key field is null, then this operation always returns null. It doesn't make sense to compare key fields in the stream to null, since null is never equivalent to any value (including null).

- **Get value by match** – Get a record from the window that matches the given record. Unlike getting a value by key, there might be more than one matching record. If there is more than one matching record, one of the matching records is returned. If there is no such match in the window, null is returned.

  Syntax: `windowName{ recordValue }`

Type: The record must be consistent with the record type of the window. The operation returns a value of the record type of the window.

Example: `input_window{ [ | d = 5 ] }`

You can use key and non-key fields in the record.

You can also iterate through all the records in a window using a "for" loop.

Examples

The following examples show complete projects that incorporate SPLASH code using the CCL Flex operator.

This project displays the top three prices for each stock symbol.

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

/* ****************************************************
 * Create a Nasdaq Trades Input Window
 */
CREATE INPUT WINDOW QTrades SCHEMA
TradesSchema PRIMARY KEY (Id)
;

/* ****************************************************
 * Use Case a:
 *          Keep records corresponding to only the top three
 * distinct values. Delete records that falls of the top
 * three values.
 *
 * Here the trades corresponding to the top three prices
 * per Symbol is maintained. It uses
 * - eventcaches
 * - local UDF
 */
CREATE FLEX Top3TradesFlex
    IN QTrades
    OUT OUTPUT WINDOW Top3Trades SCHEMA TradesSchema PRIMARY
KEY(Symbol,Price)
    BEGIN
        DECLARE
            eventCache(QTrades[Symbol], manual, Price asc)
tradesCache;
            /*
             * Inserts record into cache if in top 3 prices and
returns
             * the record to delete or just the current record if it
```

```
was
             * inserted into cache with no corresponding delete.
             */
            typeof(QTrades) insertIntoCache( typeof(QTrades)
qTrades )
            {
               // keep only the top 3 distinct prices per symbol in
the
               // event cache
               integer counter := 0;
               typeof (QTrades) rec;
               long cacheSz := cacheSize(tradesCache);
               while (counter < cacheSz) {
                   rec := getCache( tradesCache, counter );
                if( round(rec.Price,2) = round(qTrades.Price,2) ) {
                       // if the price is the same update
                       // the record.
                       deleteCache(tradesCache, counter);
                       insertCache( tradesCache, qTrades );
                       return rec;
                       break;
                   } else if( qTrades.Price < rec.Price) {
                       break;
                   }
                   counter++;
               }

               //Less than 3 distinct prices
               if(cacheSz < 3) {
                   insertCache(tradesCache, qTrades);
                   return qTrades;
               } else {    //Current price is > lowest price
                   //delete lowest price record.
                   rec := getCache(tradesCache, 0);
                   deleteCache(tradesCache, 0);
                   insertCache(tradesCache, qTrades);
                   return rec;
               }

               return null;
            }
        END;

        ON QTrades {
            keyCache( tradesCache, [Symbol=QTrades.Symbol;|] );
            typeof(QTrades) rec := insertIntoCache( QTrades );
            if(rec.Id) {
                //When id does not match current id it is a
                //record to delete
                if(rec.Id <> QTrades.Id) {
                    output setOpcode(rec, delete);
                }
                output setOpcode(QTrades, upsert);
            }
        };
    END;
```

This project collects data for thirty seconds and then computes the desired output values.

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

/* ******************************************************
 * Create a Nasdaq Trades Input Window
 */
CREATE INPUT WINDOW QTrades SCHEMA
TradesSchema PRIMARY KEY (Id)
;

/* ******************************************************
 * Use Case b:
 * Perform a computation every N seconds for records
 * arrived in the last N seconds.
 *
 * Here the Nasdaq trades data is collected for 30 seconds
 * before being released for further computation.
 */
CREATE FLEX PeriodicOutputFlex
    IN QTrades
    OUT OUTPUT WINDOW QTradesPeriodicOutput SCHEMA TradesSchema
PRIMARY KEY(Symbol,Price)
    BEGIN
        DECLARE
            dictionary(typeof(QTrades), integer) cache;
                END;
        ON QTrades {
                //Whenever a record arrives just insert into
dictionary.
                //The key of the dictionary is the key to the record.
            cache[QTrades] := 0;
        };
        EVERY 30 SECONDS {
                //Cycle through event cache and output all the rows
                //and delete the rows.
                for (rec in cache) {
                        output setOpcode(rec, upsert);
                }
                clear(cache);
        };
    END;

/**
 * Perform a computation from the periodic output.
 */
```

```
CREATE OUTPUT WINDOW QTradesSymbolStats
PRIMARY KEY DEDUCED
AS SELECT
    q.Symbol,
    MIN(q.Price)        Minprice,
    MAX(q.Price)        MaxPrice,
    sum(q.Shares * q.Price)/sum(q.Shares) Vwap,
    count(*) TotalTrades,
    sum(q.Shares) TotalVolume
FROM
    QTradesPeriodicOutput q
GROUP BY
    q.Symbol
;
```

CHAPTER 9   **SPLASH Statements**

SPLASH has statement forms for expressions, blocks, conditionals, output, "break" and "continue", "while" and "for" loops, as well as blocks of statements.

Block Statements

Statements can be a sequence of statements, wrapped in braces, with optional variable declarations.

For example:

```
{
  float d := 9.99;
  record.b := d;
}
```

You can intersperse variable declarations with statements:

```
{
  float pi := 3.14;
  print (string(pi));
  float e := 2.71;
  print (string(e));
}
```

Conditional Statements

Use conditional statements to specify an action based on whether a specific condition is true or false. Conditional statements in SPLASH use the same syntax as conditional statements in C.

For example:

```
if (record.a = 9)
  record.b := 9.99;
```

Note: You are not limited to a single statement. It is also possible to have a block of statements after the "if" condition, similar to the following example:

```
if (record.a > 9) {
  float d := record.a;
```

```
  record.b := d*5;
};
```

Conditionals may have optional "else" statements:

```
if (record.a = 9)
  record.b := 9.99;
else {
  float d := 10.9;
  record.b := d;
}
```

Control Statements

Use control statements to terminate or restart both `while` loops and `for` loops.

A `break` statement terminates the innermost loop; a `continue` statement starts the innermost loop over.

The `return` statement stops the processing and returns a value. This is most useful in SPLASH functions.

The `exit` statement stops the processing.

Expression Statements

You can turn any expression into a statement by terminating the expression with a semicolon.

For example:
```
setOpcode(input, 3);
```

Since assignments are expressions, assignments can be turned into statements in the same way. For instance, the following statement assigns a string to a variable "address":
```
address := '550 Broad Street';
```

For Loops

`For` loops iterate once over the records in a window, or the data in a vector or dictionary. To iterate multiple times over the records, use a window iterator. They ensure that the data is consistent while they are in use.

To loop over every record in an input window called "input_window":

```
for (record in input_window) {
  ...
```

```
}
```

The variable `record` is a new variable; you can use any name here. The scope is the statement or block of statements in the loop; it has no meaning outside the loop. You can also set equality criteria in searching for records with certain values of fields. For example:

```
for (record in input_window where c=10, d=11) {
  ...
}
```

This statement has the same looping behavior, except limited to the records whose `c` field is 10 and `d` field is 11. If you search on the key fields, the loop runs at most one time, but it will run extremely fast because it will use the underlying index of the stream.

To loop over the values in a vector "vec1" where `val` is any new variable:

```
for (val in vec1) {
 ...
}
```

The loop stops when the end of the vector is reached, or the value of the vector is null.

To loop over the values in a dictionary "dict1" where `key` is any new variable:

```
for (key in dict1) {
 ...
}
```

It is common, inside the loop, to use the expression `dict1[key]` to get the value held in the dictionary for that particular key.

Output Statements

The `output` statement schedules a record to be published in the output stream or window.

For example:

```
output [k = 10; | d = 20;];
```

If a Flex operator is sending output to a stream, all attempts to `output` a non-insert are rejected.

Note: You can use multiple output statements to process an event; the outputs are collected as a transaction block. Similarly, if a Flex operator receives a transaction block, the entire

transaction block is processed and all output is collected into another transaction block. This means that downstream streams, and the record data stored within the stream, are not changed until the entire event (single event or transaction block) is processed.

Print Statement

Concatenates and prints the given string arguments to standard out (stdout), which is redirected to `esp_server.log`.

Syntax

```
print (string [,...] )
```

Parameters

string	Either a string expression or a string constant.

Usage

This function concatenates the provided string expressions and prints them to standard out, which is redirected to the log file `esp_server.log`. Just like in C/C++ or Java, you can use '\n' to print a new line and '\t' to print a tab character. The output of the **print** statement is written to the log file immediately when you use the '\n' option; otherwise, it is written only when the Server shuts down.

Example

```
print('Trade Volume for Symbol', Trades.Symbol, ' is ',
string(Trades.Volume), '\n');
```

Switch Statements

The `switch` statement is a specialized form of conditional.

For instance, you can write:

```
switch(intvar*2) {
  case 0: print('case0'); break;
  case 1+1: print('case2'); break;
  default: print('default'); break;
}
```

This statement prints "case0" if the value of `intvar*2` is 0, "case2" if the value of `intvar*2` is 2, and "default" otherwise. The `default` is optional. The expression inside the parentheses `switch(...)` must be of base type, and the expressions following the `case` keyword must have the same base type.

As in C and Java, the `break` is needed to skip to the end. For instance, if you leave out the `break` after the first `case`, then this statement will print both "case0" and "case2" when `intvar*2` is 0:

```
switch(intvar*2) {
  case 0: print('case0');
  case 1+1: print('case2'); break;
  default: print('default'); break;
}
```

While Statements

While statements are a form of conditional processing. Use them to specify an action to take while a certain condition is met. While statements use the same syntax as while statements in C and are processed as loops.

For example:

```
while (not(isnull(record))) {
  record.b := record.a + record.b;
  record := getNext(record_iterator);
}
```

CHAPTER 10   **SPLASH Data Structures**

SPLASH may store and organize data in various sets of data structures designed to support specific data manipulation functions.

Records

A Record is a data structure that contains one or more columns along with a expression that determines the data type and value for the column. One or more columns in the record can be defined to be a key column. Each record also has a operation code with the default operation code being an "insert". The compiler implicitly determines the type for each of the columns based upon the type of the column expression. A record that is created can be stored in a Stream or it can be stored in a record variable with a compatible record type.

Record Event Details
A record type contains one or more column names with a basic type such string, long, date etc associated with it. One or more the columns can be identified as a key columns.

You can declare a record type inside any block of SPLASH code including Global/Local declare blocks, functions and the ON method of a Flex operator using one of the following syntax:

```
[ [columnType column; [...] [|] ]

or

[ [ columnType column; [...] | ] columnType column; [...] ]
```

Where:

columnType - is one of the basic types such as integer, long, float etc. column - the name of a column in the record. A column name must be unique within a record and is case sensitive i.e. 'symbol' is not the same as 'Symbol'.

In the previous syntax the outer square brackets is part of the syntax and does not represent an optional element. Also any columns appearing before the | character represents the key columns in the record type. Note that the semicolon following the last column before the key separator | and/or the trailing] is optional.

The following is an example:
```
[ integer TradeId; string Symbol; | integer Volume; float Price; date
TradeTime; ] traderec;
```

The previous example declares a record variable called traderec with the specified record definition that has two columns namely TradeId and Symbol and three attribute columns namely Volume, Price and TradeTime.

Record Details

You can define a record in SPLASH inside any Global/Local function and inside the ON Method of a Flex Operator. To define a record use one of the following syntax:

```
[  column = value;  [...] [|] ]
or
[  [ column = value; [...]  |  ]   column = value; [...]   ]
```

Where:

column - is the name of a column in the record. A column name must be unique within a record and is case sensitive i.e. symbol is not the same as Symbol.

value - is any expression including constant expressions. The column type is determined by the compiler based upon the type of this expression.

In the previous syntax the outer square brackets is part of the syntax and does not represent an optional element. Also any columns appearing before the | character represents the key columns in the record type. Note that the semicolon following the last column before the key separator | and/or the trailing] is optional.

When a record is created its opcode is set to 'insert' by default. You can change the operation code using the **setOpcode** function as described in the following example:

```
[ integer TradeId; string Symbol; | integer Volume; float TradePrice;
date TradeTime; ] traderec;
traderec := [ TradeId = 1; Symbol = 'SAP'; | Volume = 100; TradePrice
= 150.0; undate('2012-03-01 10:30:35'); ];
```

In the above example the record variable traderec is assigned a record object with particular values.

Operations on Records

Operations on records:

- **Get a field** – Syntax: `record.field`

  Type: The value returned has the type of the field.

  Example: `rec.data1`
- **Assign a field** – Assign a field in a record.

  Syntax: `record.field := value`

  Type: The value must be a value matching the type of the field of the record. The expression returns a record.

Example: `rec.data1 := 10`

- **Assign a Record** – It is more efficient to assign a record than to assign individual columns of a record one at a time.

  Syntax: `record := recordObject` or `record := recordVariable`.

  Examples:

  Record object assignment `outTrades := [ TradeId = 1; Symbol = 'SAP'; | Volume = 100; TradePrice = 150.0; undate('2012-03-01 10:30:35'); ];`

  Assigning one record variable to another: `outTrades := inTrades;`

  See the section on record casting rules for information on how the compiler deals with scenarios where the source record type does not exactly match the target record type.

- **`getOpcode`** – Gets the operation associated with a record. The operations are of type integer, and have the following meaning:

  - 1 means "insert"
  - 3 means "update"
  - 5 means "delete"
  - 7 means "upsert"(insert if not present, update otherwise)
  - 13 means "safe delete"(delete if present, ignore otherwise)

  Syntax: `getOpcode(record)`

  Type: The argument must be an event. The function returns an integer.

  Example: `getOpcode(input)`

- **`setOpcode`** – Sets the operation associated with a record; the legal opCodeNumber operations are listed in the above description for `getOpcode`.

  Syntax: `setOpcode(record,opCodenumber)`

  Type: The first argument must be a record, and the second an integer. The function returns the modified record.

  Example: `setOpcode(input,insert)`

Record Casting Rules
The ESP compiler does the necessary implicit casting when assigning a source record to a target record variable where the types and column names do not match exactly. This allows you to assign either a source record to a target that does not have all the columns in the target, or a source that has more columns than the target record variable type. The following casting rules are used by the compiler.

* Columns are only copied over when the source and target column names match exactly (including case).

* If the column name matches, but the column type does not, the compiler throws an error saying that you are trying to assign an expression of a wrong type.

* Columns in the source record that do not exactly match the column names and types in the target are ignored.

* The columns in the source with no matching column in the target are automatically filled in with nulls.

The following are casting examples:

```
[ integer TradeId; string Symbol; | integer Volume; float TradePrice;
date TradeTime; ] srcTrade;
[ integer TradeId; | string Symbol; integer volume; float TradeCost;
date TradeTime; ] outTrade;

outTrade := srcTrade
```

In the previous example, Volume and TradeCost are set to null because there is no corresponding target column in the source record variable srcTrade. Note that 'volume' is ignored because the case does not match.

The compiler produces a warning and ignores the source column 'Volume' and 'TradePrice' because they do not exist in the target record type.

Symbol is automatically cast as an attribute column in the target, even though it is a key column in the source.

Note: Casting is an expensive operation. So, where possible, explicitly create a source record with exactly the same type (for example, the same number of columns with the same column names and data types) as the target record variable type.

XML Values

An XML value is a value composed of XML elements and attributes, where elements can contain other XML elements or text. XML values can be created directly or built by parsing string values. XML values cannot be stored in records, but can be converted to string representation and stored in that form.

Operations on XML Values

You can declare a variable of `xml` type and assign it to XML values:

```
xml xmlVar;
```

In addition to declaring a variable for use with XML values, you can also perform the following operations:

- **xmlagg** – Aggregate a number of XML values into a single value. This can be used only in aggregate windows or with event caches (see below). Use `xmlagg` with caution; it can

consume a large amount of memory very quickly as it produces a large verbose string proportionate to the size of the contents of each group.

Syntax: `xmlagg(xml value)`

Type: The argument must be an XML value. The function returns an XML value.

Example: `xmlagg(xmlparse(stringCol))`

- **xmlconcat** – Concatenate a number of XML values into a single value.

Syntax: `xmlconcat(xml value ..., xml value)`

Type: The arguments must be XML values. The function returns an XML value.

Example: `xmlconcat(xmlparse(stringCol), xmlparse('<t/>'))`

- **xmlelement** – Create 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)`

Type: The names must adhere to these conventions:

- A name is either a sequence of alphabetic characters, digits, and underscore characters, or a sequence of any characters enclosed in double quotation marks.
- If a name is not enclosed in double quotation marks, it must begin with an alphabetic character or an underscore.
- A name cannot contain spaces unless it is enclosed in double quotation marks.
- A name cannot be a Reserved Word unless it is enclosed in double quotation marks. Reserved words are case insensitive, so for example, a name cannot be "AND" or "and" or "AnD".
- Columns cannot be named "rowid", "bigrowtime", or "rowtime".

The function returns an XML value.

Example: `xmlelement(top, xmlattributes('data' as attr1), xmlparse('<t/>'))`

- **xmlparse** – Convert a string to an XML value.

Syntax: `xmlparse(string value)`

Type: The argument must be a string value. The function returns an XML value.

Example: `xmlparse('<tag/>')`

- **xmlserialize** – Convert an XML value to a string.

Syntax: `xmlserialize(xml value)`

Type: The argument must be an XML value. The function returns a string.

Example: `xmlserialize(xmlparse('<t/>'))`

Example

```
CREATE  INPUT  WINDOW Trades
    SCHEMA (TradeId INTEGER, Symbol STRING, TradeInfo STRING)
    PRIMARY KEY (TradeId) ;

CREATE FLEX myFlex
    IN Trades
    OUT OUTPUT WINDOW TradeReport
    SCHEMA (TradeId INTEGER, TradeDesc STRING
    PRIMARY KEY (TradeId)
    outfile "output/TradeReport.out"
BEGIN
    ON Trades {
        xml u := xmlparse('<Option OptionId="8">10000</Option>');
        xml v := xmlparse(Trades.TradeInfo);
        xml w := xmlelement(Comment, xmlattributes(Trades.Symbol as
Symbol), u, v);
        v := xmlconcat(u, v, w);
        output [TradeId = Trades.TradeId; TradeDesc =
xmlserialize(v)];
    };
END;

CREATE  OUTPUT  WINDOW XmlAggregation
    SCHEMA (Symbol STRING, TradeDesc STRING)
    PRIMARY KEY DEDUCED
    outfile "output/XmlAggregation.out"
AS
    SELECT    Trades.Symbol AS Symbol
           , xmlserialize( xmlelement (  value
                                      , xmlattributes(Trades.Symbol as
Symbol)
                                      ,
xmlagg( xmlparse(Trades.TradeInfo)))) AS TradeDesc
    FROM Trades
    GROUP BY Trades.Symbol;
```

The output for the TradeReport will be:

```
<TradeReport ESP_OPS="i" TradeId="1" TradeDesc="<Option
OptionId="8">10000</Option><Transaction Price="15.4" Volume="1000"/
><Comment Symbol="EBAY"><Option OptionId="8">10000</
Option><Transaction Price="15.4" Volume="1000"/></Comment>"/>
<TradeReport ESP_OPS="i" TradeId="2" TradeDesc="<Option
OptionId="8">10000</Option><Transaction Price="5.4" Volume="2000"/
><Comment Symbol="MSFT"><Option OptionId="8">10000</
Option><Transaction Price="5.4" Volume="2000"/></Comment>"/>
<TradeReport ESP_OPS="i" TradeId="3" TradeDesc="<Option
OptionId="8">10000</Option><Transaction Price="5.8" Volume="4000"/
><Comment Symbol="MSFT"><Option OptionId="8">10000</
Option><Transaction Price="5.8" Volume="4000"/></Comment>"/>
```

The output for the XMLAggregation will be:

```
<XmlAggregation ESP_OPS="i" Symbol="EBAY" TradeDesc="<value
Symbol="EBAY"><Transaction Price="15.4" Volume="1000"/></value>"/>
<XmlAggregation ESP_OPS="i" Symbol="MSFT" TradeDesc="<value
Symbol="MSFT"><Transaction Price="5.4" Volume="2000"/></value>"/>
<XmlAggregation ESP_OPS="u" Symbol="MSFT" TradeDesc="<value
Symbol="MSFT"><Transaction Price="5.8" Volume="4000"/><Transaction
Price="5.4" Volume="2000"/></value>"/>
```

Vectors

A vector is a sequence of values, all of which must have the same type, with an ability to access elements of the sequence by an integer index. A vector has a size, from a minimum of 0 to a maximum of 2 billion entries.

Semantics and Operations
Vectors use semantics inherited from C: when accessing elements by index, the first position in the vector is index 0.

You can declare vectors in Global or Local blocks via the syntax:

```
vector(valueType) variable;
```

For instance, you can declare a vector holding 32-bit integers:

```
vector(integer) pos;
```

You can perform the following operations on vectors:

- **Create** – Create a new empty vector.

  Syntax: `new vector(type)`

  Type: A vector of the declared type is returned.

  Example: `pos := new vector(integer);`

- **Get value by index** – Get a value from the vector. If the index is less than 0 or greater than or equal to the size of the vector, return null.

  Syntax: `vector[index]`

  Type: The index must have type integer. The value returned has the type of the values held in the vector.

  Example: `pos[10]`

- **Assign a value** – Assign a cell in the vector.

  Syntax: `vector[index] := value`

  Type: The index must have type integer, and the value must match the value type of the vector. The value returned is the updated vector.

  Example: `pos[5] := 3`

- **Determine the size** – Returns the number of elements in the vector.

  Syntax: `size(vector)`

  Type: The argument must be a vector. The value returned has type integer.

  Example: `size(pos)`
- **Insert an element** – Inserts an element at the end of the vector and returns the modified vector.

  Syntax: `push_back(vector, value)`

  Type: The second argument must be a value with the value type of the vector. The return value has the type of the vector.

  Example: `push_back(pos, 3)`
- **Change the size** – Resize a vector, either removing elements if the vector shrinks, or adding null elements if the vector expands.

  Syntax: `resize(vector, newsize)`

  Type: The second argument must have type integer. The return value has the type of the vector.

  Example: `resize(vec1, 2)`

There is no command to copy a vector. Therefore, the only way to make a copy of a vector is manually, by iterating through the elements. You can also iterate through all the elements in the vector (up to the first null element) using a **for** loop.

While dictionaries and vectors can be defined globally, design and use global structures with care. Recall that ESP is multi-threaded. Therefore, when accessing a structure that can be modified from multiple threads, be aware that the state of the structure when you are accessing it may not actually be what you assume it to be. Also consider the impact on performance, particularly when iterating over data structures. As a CCL query or flex operator iterates over a data structure, the query or flex operator locks the structure, blocking other threads. Thus, performance degrades significantly with the number of concurrent queries iterating over the global structure.

Global use of these data structures should be limited to relatively static data (such as country codes) that will not need to be updated during processing, but will be read by multiple streams. Writing the data to the dictionary or vector must be completed before any streams read it.

All operations that read a global dictionary or vector should perform an isnull check, as shown in this example.

```
>typeof(streamname) rec := dict[symbol];
if( not (isnull(rec)) {
// use rec
}
```

Dictionaries

Dictionaries are data structures that associate keys with values; like maps in C++ and Java, arrays in AWK, and association lists in LISP.

Declare a dictionary in a Global or Local block using the syntax:

```
dictionary(keyType, valueType) variable;
```

For instance, if you have an input stream called "input_stream", you could store an integer for distinct records as

```
dictionary(typeof(input_stream), integer) counter;
```

Only one value is stored per key. It is therefore important to understand what equality on keys means. For the simple datatypes, equality means the usual equality, for example, equality on integer or on string values. For record types, equality means that the keys match (the data fields and operation are ignored).

Dictionaries can be defined anywhere that a variable can be defined: globally or locally

While dictionaries and vectors can be defined globally, design and use global structures with care. Recall that ESP is multi-threaded. Therefore, when accessing a structure that can be modified from multiple threads, be aware that the state of the structure when you are accessing it may not actually be what you assume it to be. Also consider the impact on performance, particularly when iterating over data structures. As a CCL query or flex operator iterates over a data structure, the query or flex operator locks the structure, blocking other threads. Thus, performance degrades significantly with the number of concurrent queries iterating over the global structure.

Global use of these data structures should be limited to relatively static data (such as country codes) that will not need to be updated during processing, but will be read by multiple streams. Writing the data to the dictionary or vector must be completed before any streams read it.

When you create a dictionary, you can specify a `valueType` of dictionary to create a dictionary of dictionaries. This structure is useful when your logic involves separating data into two levels (for example, a dictionary of telephone area codes containing dictionaries of the exchanges within each area code). But, this structure requires a lookup at each level, and lookups are time-consuming.

Operations on Dictionaries

Dictionaries are data structures that associate keys with values. You can perform specific operations on dictionaries.

Note: All operations that read a global dictionary or vector should perform an isnull check, as shown in this example.

```
>typeof(streamname) rec := dict[symbol];
if( not (isnull(rec)) {
// use rec
}
```

You can perform the following operations on dictionaries:

- Create a new dictionary. Memory is allocated when the dictionary is created, even though it is empty.

  Syntax: `new dictionary(type_of_key, type_of_value)`

  For example, `d := new dictionary(integer, string);` creates a new, empty dictionary with integer keys and string values.
- Assign a value to a key in the dictionary. The key and value must match the key type and value type of the dictionary. The function returns the updated dictionary.

  Syntax: `dictionary[key] := value`

  For example, `counter[input] := 3` returns the dictionary counter with the value of input set to 3.
- Get a value from the dictionary by key. The key must have the type of the keys of the dictionary. This operator returns a value of the type of the values held in the dictionary. Unless the key is not found in the dictionary, then it returns null.

  Syntax: `dictionary[key]`

  For example, `counter[input]` returns the value associated with the key input.
- Remove a key, and its associated value, from the dictionary. The key must match the key type of the dictionary. The function returns an integer: 0 if the key was not present, and 1 otherwise.

  Syntax: `remove(dictionary, key)`

  For example, `remove(counter, input)` returns a 1 after removing the key input and its associated value, or it returns a 0 if the key input was not found.
- Remove all key/value pairs from the dictionary. Each key/value pair is examined in turn; if not referenced anywhere else, it is removed and memory is deallocated. The function returns the cleared dictionary.

  Syntax: `clear(dictionary)`

  For example, `clear(counter)` returns an empty dictionary, counter, after removing all of the key/value pairs as long as none of them were referenced anywhere else.
- Test a dictionary for emptiness. The function returns an integer: 1 if the dictionary is empty, 0 if not empty.

  Syntax: `empty(dictionary)`

  For example, `empty(counter)` returns a 1 if counter is empty, a 0 if it is not.

There is no command to copy a dictionary. Therefore, the only way to make a copy of a dictionary is manually, by iterating through the elements. You can also iterate through all the elements in the dictionary (up to the first null element) using a **for** loop.

Window Iterators

Window iterators are a means of explicitly iterating over all of the records stored in a window. It is usually more convenient, and safer, to use the `for` loop mechanism if the goal is to iterate over the data once, but iterators provide extra flexibility.

Functions for Iterators

Each block of code has implicit variables for windows and window iterators. If an input window is named `Window1`, there are variables `Window1_stream` and `Window1_iterator`.

Those variables can be used in conjunction with the following functions.

- **deleteIterator** – Releases the resources associated with an iterator.

  Syntax: `deleteIterator(iterator)`

  Type: The argument must be an iterator expression. The function returns a null value.

  Example: `deleteIterator(input1_iterator)`

  Note: Iterators are not implicitly deleted. If you don't delete them explicitly, all further updates to the stream may be blocked.

- **getIterator** – Get an iterator for a window.

  Syntax: `getIterator(windowName)`

  Type: The argument must be a window expression. The function returns an iterator.

  Example: `getIterator(Window1)`

- **getNext** – Returns the next record in the iterator, or null if there are no more records.

  Syntax: `getNext(iterator)`

  Type: The first argument must be an iterator expression. The function returns a record, or "null" if there is no more data in the iterator.

  Example: `getNext(input_iterator)`

- **resetIterator** – Resets the iterator to the beginning.

  Syntax: `resetIterator(iterator)`

  Type: The argument must be an iterator expression. The function returns an iterator.

  Example: `resetIterator(input_iterator)`

- **setRange** – Sets a range of columns to search for. Subsequent getNext calls return only those records whose columns match the given values.

  Syntax: `setRange(iterator fieldName... expr...)`

Type: The first argument must be an iterator expression; the next arguments must be the names of fields within the record; the final arguments must be expressions. The function returns an iterator.

Example: `setRange(input_iterator,Currency,Rate,'EUR',9.888)`

* **setSearch** – Sets values of columns to search for. Subsequent getNext calls return only those records whose columns match the given values.

Syntax: `setSearch( iterator number... expr...)`

Type: The first argument must be an iterator expression; the next arguments must be column numbers (starting from 0) in the record; the final arguments must be expressions. The function returns an iterator.

Example: `setSearch(input_iterator,0,2,'EUR',9.888)`

Note: The `setSearch` function has been deprecated because it requires a specific layout of fields. It has been retained for backwards compatibility with existing projects. When developing new projects, use the `setRange` function instead.

Event Caches

Event caches are alternate windowing mechanisms and data structures. They are organized into buckets, with each bucket able to store events or records and each bucket based on values of the fields in the records. Event caches are often used when vectors or dictionaries are not quite the right data structure, such as when a window-type store is needed within a FLEX operator, or as an alternative to the CCL KEEP clause when greater control or flexibility is required.

You can define an event cache in a Local block. A simple event cache declaration:

```
eventCache(input_stream) e0;
```

This event cache holds all the events for an input stream "input_stream". The default key structure of windows define the bucket policy. That is, the buckets in this stream correspond to the keys of the input stream. When the input of an event cache is a window or delta stream, the default bucket policy is set to the primary key of the window or delta stream. When the input of an event cache is an insert-only stream, there is no default bucket policy and a single bucket is created for all the events. However, because streams have no keys, the default behavior is for all the rows in the streams to go into one bucket in the event cache.

Suppose the input stream in this case has two fields, a key field `k` and a data field `d`. Suppose the events have been:

```
<input_stream ESP_OPS="i" k="1" d="10"/>
<input_stream ESP_OPS="u" k="1" d="11"/>
<input_stream ESP_OPS="i" k="2" d="21"/>
```

After these events have flowed in, there will be two buckets. The first bucket will contain the first two events, because these have the same key; the second bucket will contain the last event.

Event caches allow for aggregation over events. That is, the ordinary aggregation operations that can be used in aggregate windows can be used in the same way over event caches. The "group" that is selected for aggregation is the one associated with the current event (i.e. the event that has just arrived).

```
<input_stream ESP_OPS="u" k="1" d="12"/>
```

For instance, if the above event appears in this stream, then the expression `sum(e0.d)` returns 10+11+12=33. You can use any of the accepted aggregation functions, including `avg`, `count`, `max`, and `min`.

Manual Insertion

By default, every event that comes into a stream with an event cache gets put into the event cache.

You can explicitly indicate this default behavior with the `auto` option:

```
eventCache(instream, auto) e0;
```

You can also put events into an event cache if they are marked `manual`:

```
eventCache(instream, manual) e0;
```

Use the function `insertCache` to do this.

Changing Buckets

An event cache organizes events into buckets. By default, the buckets are determined from the keys of the input stream/window. You can change that default behavior to alternative keys, specifying other fields in square brackets after the name of the input.

Specifying the following keeps buckets organized by distinct values of the `d0` and `d1` fields:

```
eventCache(instream[d0,d1]) e0;
```

To keep one large bucket of all events, write the following:

```
eventCache(instream[]) e0;
```

Managing Bucket Size

You can manage the size of buckets in an event cache. That can often be important in controlling the use of memory.

You can limit the size of a bucket to the most recent events, by number of seconds, or by time:

```
eventCache(instream, 3 events) e0;
eventCache(instream, 3 seconds) e1;
```

You can also specify whether to completely clear the bucket when the size or time expires by specifying the jump option:

```
eventCache(instream, 3 seconds, jump);
```

The default is nojump.

All of these options can be used together. For example, this example clears out a bucket when it reaches 10 events (when the 11th event comes in) or when 3 seconds elapse.

```
eventCache(instream, 10 events, 3 seconds, jump);
```

Keeping Records

You can keep records in an event cache, instead of distinct events for insert, update, and delete, by specifying the coalesce option.

For example:

```
eventCache(instream, coalesce) e0;
```

This option is most often used in conjunction with the ordering option.

Ordering

Normally, the events in a bucket are kept by order of arrival. You can specify a different ordering by the fields of the events.

For instance, to keep the events in the bucket ordered by field d in descending order:

```
eventCache(instream, d desc) e0;
```

You can order by more than one field. The following example orders the buckets by field d0 in descending order, then by field d1 in ascending order in case the d0 fields are equal.

```
eventCache(instream, d0 desc, d1 asc) e0;
```

Operations on Event Caches

Event caches hold a number of previous events for the input stream(s)/window(s).

Supported Event Cache Operations

- **expireCache** – Remove events from the current bucket that are older than a certain number of seconds.

  Syntax: expireCache(events, seconds)

  Type: The first argument must name an event cache variable. The second argument must be an integer. The function returns the event cache.

  Example: expireCache(events, 50)

- **insertCache** – Insert a record value into an event cache.

  Syntax: insertCache(events, record)

  Type: The first argument must name an event cache variable. The argument must be a record type. The function returns the record inserted.

  Example: insertCache(events, inputStream)

- **keyCache** – Select the current bucket in an event cache. Normally, the current input record selects the active bucket. You might want to change the current active bucket in some cases. For example, during the evaluation of the debugging expressions, there is no current input record and thus no bucket is set by default. The only way to set the bucket then is to do it manually using this function.

  Syntax: keyCache(events, event)

  Type: The first argument must name an event cache variable. The second argument must be a record type. The function returns the same record.

  Example: keyCache(ec1, rec)

- **getCache** – Returns the row specified by a given index from the current bucket in the event cache. This index is 0 based. The function takes an integer as its argument, and the function returns a row. Specifying an invalid index parameter will result in the generation of a bad record.

  Syntax: getCache(cacheName, index)

  Type: The first argument must name an event cache variable. The second argument must be an integer specifying the row to retrieve. The function returns the specified row of the cache.

  Example: getCache(tradesCache, 3)

- **deleteCache** – Returns the row specified by a given index from the current bucket in the event cache. This index is 0 based. The function takes an integer as its argument, and the function returns a row. Specifying an invalid index parameter will result in the generation of a bad record.

  Syntax: deleteCache(cacheName, index)

  Type: The first argument must name an event cache variable. The second argument must be an integer specifying the row to delete. The function deletes the specified row; it does not return any output.

  Example: deleteCache(tradesCache, 0)

- **cacheSize** – Returns the size of the current bucket in the event cache.

  Syntax: cacheSize(cacheName)

  Type: This function takes an argument of the name of the event cache variable. It then returns a long.

  Example: cacheSize(tradesCache)

APPENDIX A   **List of Keywords**

Reserved words in CCL that are case-insensitive. Keywords cannot be used as identifiers for any CCL objects.

A list of keywords present in CCL:

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	hash	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)
partition	partitions				
pattern	primary	properties	rank	records	retain
return	right	roundrobin	row(s)	safedelete	schema
sec					
second(s)	select	set	setRange	slack	start

APPENDIX A: List of Keywords

static	store(s)	stream	sum	sync	switch
then	times	to	top	transaction	true
type	typedef	typeof	union	update	upsert
values	when	where	while	window	within
xmlattri-butes	xmlelement				

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 SAP 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/Buenos_Aires
America/Argentina/Catamarca	America/Argentina/ComodRivadavia	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/Ushuaia	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	CAT	CET
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.

Event Stream Processor Time Formatting Codes

Column Code	Description	Input	Output
MM	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 characters (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

Column Code	Description	Input	Output
DD	Day of month (1-31).	Y	Y
DDD	Day of year (1-366).	N	Y
DAY	Name of day (SUNDAY, MONDAY, ..., SATURDAY).	Y	Y
DY	Abbreviated name of day (SUN, MON, ..., SAT).	Y	Y
HH	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
PM	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 format, 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 digits 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 precede 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	N
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
TZH	Time zone hour displacement. For example, -5 indicates a time zone five hours earlier than GMT.	N	Y
TZM	Time zone hour and minute displacement. For example, -5:30 indicates a time zone that is five hours and 30 minutes earlier than GMT.	N	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.

The Event Stream Processor supports the following strftime() codes:

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 between 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 between 0 and 6, with Sunday represented as 0.

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 %.

See also
*   *to_bigdatetime()* on page 134

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 Sunday=6.
weekendEnd	An integer that represents a day of the week, when Monday=0, Tuesday=1, ..., Saturday=5, and Sunday=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
```

APPENDIX B: Date and Time Programming

APPENDIX C **Statement on Support for Multibyte Characters**

SAP Sybase Event Stream Processor supports UTF-8 encoded data within data streams, but with some limitations.

1. UTF-8 encoded data is supported in both input streams and derived streams (including output streams). Thus, events streamed or loaded into Source Streams may contain UTF-8 encoded data, and this data is correctly carried through the project. Testing has shown that the server and studio are able to receive, store, display and output UTF-8 encoded data.
2. String functions support non-ASCII data when the **utf8** project deployment option in the project configuration (CCR) file is set to *true*. The only operators supported for non-ASCII UTF-8 strings are =, <, >. The use of non-ASCII string data in expressions in any other way (including filter expressions) is not supported. For information on the project configuration file, see the *Configuration and Administration Guide* and the *Studio Users Guide*.
3. Constants and literals cannot be assigned UTF-8 values outside the ASCII range.
4. Adapters have not been tested with (non-ASCII) UTF-8 data.
5. Non-ASCII characters are not supported in metadata such as stream names, column names, and so on.
6. The Studio interface, error messages, logs, and so on are only supported in English.

Index

G

X

Y