Programmers Guide

SAP Sybase Event Stream Processor 5.1 SP04

DOCUMENT ID: DC01612-01-0514-01
LAST REVISED: November 2013

Contents

CHAPTER 1: Introduction 1

Data-Flow Programming 1

Continuous Computation Language 2

SPLASH 3

Authoring Methods 3

CHAPTER 2: CCL Project Basics 5

Events 5

Operation Codes 6

Streams 7

Windows 7

Retention 8

Named Windows 11

Unnamed Windows 11

Delta Streams 13

Comparing Streams, Windows, and Delta Streams 14

Bindings on Streams, Delta Streams, and Windows 15

Input/Output/Local 19

Implicit Columns 20

Schemas 21

Stores 21

CCL Continuous Queries 22

Adapters 24

Order of Elements 24

CHAPTER 3: Developing a Project in CCL 25

CHAPTER 4: CCL Language Components 27

Datatypes **27**
Intervals 31
Operators **32**
Expressions **35**
CCL Comments **37**
Case-Sensitivity **38**
Literals **39**
Time Literals 39
Boolean Literals 40
String Literals 40
Numeric Literals 41

CHAPTER 5: CCL Query Construction **43**

Filtering **43**
Splitting Up Incoming Data **43**
Unions **45**
Example: Merging Data from Streams or Windows 45
Joins **46**
Key Field Rules 48
Join Examples: ANSI Syntax 49
Join Example: Comma-Separated Syntax 51
Pattern Matching **51**
Aggregation **52**
Reference Table Queries **54**
Using a Reference Table Query in a Join 55
Using a Reference Table Query in a Module 57
Using a Reference Table Query in SPLASH 58
Using a Reference Table Query in SPLASH to Get a Single Row from a Table 60

CHAPTER 6: Advanced CCL Programming Techniques **63**

Declare Blocks **63**
Typedefs 64

Parameters 64
Variables 65
Declaring Project Variables, Parameters, Datatypes, and Functions 67
Flex Operators 68
Automatic Partitioning 68
Guidelines for Partitioning Flex Operators 72
Guidelines for Partitioning SPLASH 75
Guidelines for Partitioning Elements with Retention Policies 78
Guidelines for Partitioning Aggregation 80
Guidelines for Partitioning Modules 81
Guidelines for Partitioning Joins 83
Guidelines for Partitioning Elements on Log Stores 85
Modularity 85
Module Creation and Usage 86
Example: Creating and Using Modules 86
Example: Parameters in Modules 87
Data Recovery 89
Log Store Optimization Techniques 90
Error Streams 90
Monitoring Streams for Errors 92
CHAPTER 7: Zero Data Loss 95
Consistent Recovery 95
Auto Checkpoint 96
Guaranteed Delivery 98
Achieving Zero Data Loss 100
CHAPTER 8: Creating a Log Store 103
Log Store Guidelines 104
Sizing a Log Store 105
Log Store Sizing Reference 109

CHAPTER 9: Writing SPLASH Routines 113

Internal Pulsing 113
Order Book 114

CHAPTER 10: Integrating SPLASH into CCL 117

Access to the Event 117
Access to Input Windows 117
Output Statement 119
Notes on Transactions 119

CHAPTER 11: Using SPLASH in Projects 121

CHAPTER 12: ESP Datatypes in SQL Queries 125

CHAPTER 13: PowerDesigner for Event Stream Processor 127

Getting Started 127
Data Modeling Scenarios 127
Sample PowerDesigner Project 128
Opening the Sample Project 128
Learning More About PowerDesigner 128
Data Model 129
ESP Schema Logical Data Model 129
Finding an Object in a Diagram 129
Data Model Tables 129
Extensions 131
Category Set 132
Schema Definitions 132
Impact and Lineage Analysis 133
Extended Model Setup 134
Extending an Existing Model 134

Setting Up the Model Category Set File 134

Merging ESP Categories 135

Changing the Default Category 135

Setting Datatypes for an ESP Schema 135

ESP Schema Model Development 136

Exploring the Sample Model 137

The Sample Model 137

Creating an ESP Schema Model 139

Creating a Model Using Categories 139

Creating a Logical Data Model 139

Adding Schema Definition 140

Defining Schema Properties 141

Validating a Model 144

PowerDesigner Validity Checks 145

Custom Checks for ESP Schema Extensions ...145

Importing a CCL File 146

Exporting a CCL File 146

Model Generation 146

Generating a new Sybase IQ, SAP HANA, or ASE Model from an ESP Schema Model 147

Checking Indexes 148

Setting Physical Options 148

Adding Foreign Keys 149

Generating a new ESP Schema Model from a Sybase IQ, SAP HANA, or ASE Model 149

Updating an existing Sybase IQ, SAP HANA, or ASE Model from an ESP Schema Model 149

Updating an existing ESP Schema Model from a Sybase IQ, SAP HANA, or ASE Model 150

Impact and Lineage Analysis 150

Launching an Impact and Lineage Analysis 151

Generating an Analysis Diagram 152

Reviewing an Impact and Lineage Analysis 152

Sample Analysis for a Schema Definition 153

Sample Analysis for a Table 154

DDL Script Generation 155
Generating Database Schema with PowerDesigner ...155
Changing the Default Database User 155
Generating DDL Scripts 156
Executing DDL Scripts for the SAP Sybase IQ Database 157
Executing DDL Scripts for the SAP HANA Database 157
Executing DDL Scripts for the ASE Database 158

APPENDIX A: List of Keywords 159

APPENDIX B: Date and Time Programming 161
Time Zones 161
Changes to Time Zone Defaults 162
List of Time Zones 162
Date/Time Format Codes 170
Calendar Files 174

APPENDIX C: Performance and Tuning Tips 177
Distributing Load through Parallelization 177
Distributing Load through Modularization 180
Data Flow in Event Stream Processor 181
Log Store Considerations 181
Batch Processing 182
Main Memory Usage 182
Determining Stream Memory Usage 183
CPU Usage 184
TCP Buffer and Window Sizes 185
Improving Aggregation Performance 185

Index 189

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

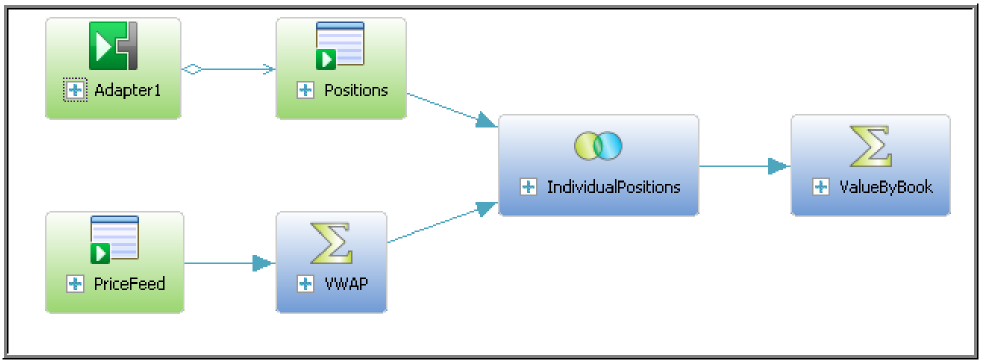
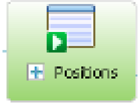
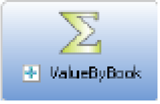


Table 1. Data-Flow Diagram Contents

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.

Events

A business event is a message that contains information about an actual business event that occurred. Many business systems produce streams of such events as things happen.

You can use streams, windows, and delta streams with adapters to create complex projects. Streams, windows, and delta streams allow you to consume and process input events and generate output events.

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

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

Event Blocks

Business events can be published into an ESP model in collections called Event Blocks, improving the performance of your ESP model. Event blocks come in two different types: envelopes and transactions. As an event block is being processed by a window, resulting rows are not sent downstream immediately. Instead, they are stored until the last event of the block is processed, and the resulting events are then sent downstream. Event blocks have the following properties:

- Envelopes:
  - Each row in an envelope is treated atomically; a failure in an event does not discard the envelope. This behavior is useful if a model's performance is important, but not necessarily the integrity of the data.
- Transactions:

- A transaction will be discarded if any one event in the block fails. This behavior can be used to guarantee that logical blocks of events are completely error-free.
- Before a transaction block is sent downstream, all events in the transaction are compressed as much as possible. For example, an event with an insert and then an update will compress down to a single insert with updated values.

Operation Codes

The operation code (opcode) of an event record specifies the action to perform on the underlying store of a window for that event.

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

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

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

To handle information sets that are updated by incoming events, Event Stream Processor recognizes the following opcodes in incoming event records:

- **insert –** Insert the event record.
- **update –** Update the record with the specified key. If no such record exists, it is a runtime error.
- **delete –** Delete the record with the specified key. If no such record exists, it is a runtime error.
- **upsert –** If a record with a matching key exists, update it. If a record with a matching key does not exist, insert this record.
- **safedelete –** If a record with a matching key exists, delete it. If a record with a matching key does not exist, do nothing.

All event records include an opcode. Each stream or window in the project accepts incoming event records and outputs event records. Output events, including opcodes, are determined by their source (stream, window, or delta stream) and the processing specified for it.

Refer to the *Streams*, *Windows*, and *Delta Streams* topics in the *Programmers Guide* for details on how each interprets the opcodes on incoming event records and generates opcodes for output records.

Streams

Streams subscribe to incoming events and process the event data according to the rules you specify (which can be thought of as a "continuous query") to publish output events. Because they are stateless, they cannot retain data.

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

Because a stream does not have an underlying store, the only thing it can do with arriving input events is insert them. Insert, update, and upsert opcodes are all treated as inserts. Delete and safedelete are ignored. The only opcode that a stream can include in output event records is insert.

Local and output streams take their input from other streams or windows, rather than from adapters, and they apply a continuous query to produce their output. Local streams are identical to output streams, except that local streams are hidden from outside subscribers. Thus, a subscriber cannot subscribe to a local stream. You cannot create primary keys for streams. You cannot monitor or subscribe to local streams in ESP Studio.

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.

Unnamed windows are implicitly created in two situations: when using a join with a window that produces a stream, and when the **KEEP** clause is used with the **FROM** clause of a statement. In both situations, when an unnamed window is created it always includes a primary key.

Note: The unnamed window uses additional memory; there is no memory reporting on unnamed windows.

This example creates an unnamed window when using a join with a window:

```
CREATE INPUT WINDOW Win1 SCHEMA (Key1 INTEGER, Col1 STRING, Col2
STRING) PRIMARY KEY (Key1);
CREATE INPUT WINDOW Win2 SCHEMA (Key1 STRING, Col3 STRING) PRIMARY
KEY (Key1);

CREATE OUTPUT WINDOW Out1 PRIMARY KEY DEDUCED AS SELECT Win1.Key1,
Win1.Col1, Win1.Col2, Win.2.Col3
FROM Win1 INNER JOIN Win2 ON Win1.Col1 = Win2.Key1;
```

Note: The unnamed window is created to ensure that a join does not see records that have not yet arrived at the join. This can happen because the source to the join and the join itself are running in separate threads.

The following three examples demonstrate when an unnamed window is created using the **KEEP** clause:

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. The 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.

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

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
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 **insert**/**update**/**delete**	Produces insert. Converts update to insert. Ignores delete.	Produces insert, update, or delete according to the exceptions listed below. Generates an error on duplicate inserts, bad updates, and bad deletes. Windows with a retention policy treat update as upsert.	Produces insert, update, or delete according to the exceptions listed below. Does not detect duplicate inserts, bad updates, or bad deletes. Delta streams produced by a Flex Operator do not receive updates.

Feature Capability	Streams	Windows	Delta Streams
Behavior on receiving **upsert**/**safedelete**	Converts upsert to insert. Ignores safedelete.	Produces insert, update, or delete according to the exceptions listed below.	Delta streams do not receive upserts or safedeletes.

Exceptions

A window can produce output event records with different opcodes depending on what changes are made to the contents of its store by the incoming event and its opcode. For example:

- In a window performing aggregation logic, an incoming event record with an insert opcode can update the contents of the store and output an event record with an update opcode. This can also happen in a window implementing a left join.
- In a window with a count-based retention policy, an incoming event record with an insert opcode can cause the store to exceed this count. The window deletes the excess rows, producing an event record with a delete opcode.

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. If it does not meet the criteria, no opcode is 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. If the original insert meets the filter criteria but the update does not, it outputs a delete opcode.
- An input record with a delete opcode outputs with a delete opcode, as long as it meets the filter criteria.

Bindings on Streams, Delta Streams, and Windows

Bindings enable data to flow between projects. When you create a binding, a stream, delta stream, or window in one project subscribes or publishes to a stream, delta stream, or window in another project.

A binding is a named connection from an input or output stream (or delta stream or window) of one project to an input stream (or delta stream or window) of another; you can configure it at either end.

- An input stream can subscribe to one or more streams in other projects. The stream subscribed to need not be an output stream—you can create an output binding on an input stream. For more information, see *Example: Configuring an Output Binding on an Input Stream or Window* on page 17.

- An output stream can publish to one or more input streams in other projects. An output stream cannot receive incoming data, whether by subscription or publication.

Bindings reside in the CCR project configuration file so you can change them at runtime. The streams being bound must have compatible schemas.

Example: Binding to a Stream on an SSL-Enabled Cluster

This example shows a binding called BaseInputBinding that connects a local input stream called sin to a remote output stream that is also called sin. When the SSL protocol is enabled on the manager node of the data source stream's cluster, the <Manager> element that specifies the cluster hostname and port in the CCR file must include the https:// prefix, as shown here. If you omit the https:// prefix, the binding cannot pass data, so the input stream will not receive anything.

```
<Configuration>
  <Runtime>
    <Clusters>
      <Cluster name="cluster1" type="remote">
        <Username>USER_NAME</Username>
        <Password>PASSWORD</Password>
        <Managers>
          <Manager>https://CLUSTER_MANAGER_HOSTNAME:
          CLUSTER_MANAGER_RPC_PORT</Manager>
          <!-- use https:// when SSL is enabled -->
        </Managers>
      </Cluster>
    </Clusters>

    <Bindings>
      <Binding name="sin">
        <Cluster>cluster1</Cluster>
        <Workspace>ws2</Workspace>
        <Project>prj2</Project>
        <BindingName>BaseInputBinding</BindingName>
        <RemoteStream>sin</RemoteStream>
      </Binding>
    </Bindings>
  </Runtime>
</Configuration>
```

Example: Reconnection Intervals for Bindings

This example shows two bindings, b1 and b2, on a local input stream called MyInStream. The b1 binding includes a reconnection interval option specifying that if the connection between MyInStream and the remote output stream is lost, the project will attempt to reconnect every 10 seconds. Because the b2 binding does not specify a reconnection interval, its reconnection attempts will occur at the default interval of five seconds. To suppress all reconnection attempts, set <ReconnectInterval> to 0. Use positive whole number values to set the reconnection interval in seconds.

```
<Bindings>
  <Binding name="MyInStream">
    <Cluster>c1</Cluster>
    <Workspace>w1</Workspace>
    <Project>p1</Project>
    <BindingName>b1</BindingName>
    <RemoteStream>MyInStream1</RemoteStream>
    <ReconnectInterval>10</ReconnectInterval>
  </Binding>
  <Binding name="MyInStream">
    <Cluster>c1</Cluster>
    <Workspace>w1</Workspace>
    <Project>p1</Project>
    <BindingName>b2</BindingName>
    <RemoteStream>MyInStream2</RemoteStream>
  </Binding>
</Bindings>
```

Example: Configuring an Input Stream or Window to Provide Output

This example shows how to configure an input stream to send data to another input stream by setting the <Output> parameter in the <Binding> element to true.

Note: Set the <Output> parameter to true only when you configure a binding on an input stream or window that is providing output. If you configure the binding on the stream or window that is receiving input, do not set the <Output> parameter. (It is never necessary to set the <Output> parameter when you configure a binding on an output stream; output streams can only produce output.)

In this example, output from the input stream MyInStream, in the local project, is bound to the input stream MyInStream1 in project p2. The line `<Output>true</Output>` tells the binding to publish (send data out) to the remote stream. Without that line, this binding would subscribe to data from MyInStream1 because bindings on input streams receive data by default.

```
<Binding name="MyInStream">
  <Cluster>c1</Cluster>
  <Workspace>w1</Workspace>
  <Project>p2</Project>
  <BindingName>b1</BindingName>
  <RemoteStream>MyInStream1</RemoteStream>
  <Output>true</Output>
</Binding>
```

Example: Configuring a Window for Guaranteed Delivery

This example shows how to enable and configure guaranteed delivery (GD) on a window’s output binding. The GD parameters are the same for input bindings.

Enable GD for a binding to guarantee that if the connection between the binding and the remote stream is severed (by shutting down the project that contains the local stream, for example), all transactions that are supposed to be transmitted through the binding during its downtime are processed once the connection is re-established.

Use these parameters in the <Binding> element of your CCR file to set a binding to support guaranteed delivery:

- <EnableGD> – Specifies whether guaranteed delivery is enabled for this binding. Values are true and false.

  Note: When you enable GD on a binding, make sure:

  - The binding's source data window is running in GD mode or GD mode with checkpoint.
  - The binding's target data window is backed by a log store.

- <GDName> – Supply a unique name for the GD session (subscription) this binding establishes.
- <GDBatchSize> – The number of transactions this binding may collect in a batch before releasing the batch to the target window. The binding issues a **GD commit** to the source data window after releasing the data. This setting is ignored when the source data window is in GD mode with checkpoint and the <EnableGDCache> parameter on this binding is set to true. Default is 10.
- <EnableGDCache> – Enable this binding to cache data. When the source data window is in GD mode with checkpoint, the binding receives checkpoint messages indicating the last row of data that has been checkpointed by the window. If the binding is enabled for GD caching, it caches incoming transactions until it receives a checkpoint message from the source window. The checkpoint message triggers the binding to send to the target window all cached transactions up to the one indicated in the checkpoint message. The binding issues a **GD commit** to the source data window after releasing cached data. If GD caching is disabled, checkpoint messages are ignored and the binding forwards data based on the value of <GDBatchSize>. The setting of <EnableGDCache> is ignored if the source data window is not in GD mode with checkpoint. Values are true and false; default is true.

In this example, output from the local output stream MyOutStream is bound to MyInStream1 in project p1. GD and GD caching are enabled. The GD session name is b1_GD1 and the GD batch size is 20 transactions.

```
<Binding name="MyOutStream">
  <Cluster>c1</Cluster>
  <Workspace>w1</Workspace>
  <Project>p1</Project>
  <BindingName>b1</BindingName>
  <RemoteStream>MyInStream1</RemoteStream>
  <ReconnectInterval>5</ReconnectInterval>
  <EnableGD>true</EnableGD>
  <GDName>b1_GD1</GDName >
  <GDBatchSize>20</GDBatchSize >
  <EnableGDCache>true</EnableGDCache >
</Binding>
```

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).

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 can use the same schema, but each 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 LOG STORE** or **CREATE DEFAULT MEMORY STORE** statements, 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 LOG 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 todefine 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
[...]
```

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 Developing a Project in CCL

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

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

1. Create a `.ccl` file.

   Creating a project in SAP Sybase Event Stream Processor Studio creates the `.ccl` file automatically.
2. Add input streams and windows.
3. Add output streams and windows with simple continuous queries.
4. Attach adapters to streams and windows to subscribe to external sources or publish output.
5. Compile the CCL code.
6. Run the compiled project against test data, using the debugging tools in SAP Sybase Event Stream Processor Studio and command line utilities.

   Repeat this step as often as needed.
7. Add queries to the project. Start with simple queries and gradually add complexity.
8. (Optional) Use functions in your continuous queries to perform mathematical operations, aggregations, datatype conversions, and other common tasks:

   - Built-in functions for many common operations
   - User-defined functions written in the SPLASH programming language
   - User-defined external functions written in C/C++ or Java
9. (Optional) Create named schemas to define a reusable data structure for streams or windows.
10. (Optional) Create memory stores or log stores to retain the state of data windows in memory or on disk.
11. (Optional) Create modules to contain reusable CCL that can be loaded multiple times in a project.

CHAPTER 4 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
`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.
`bigint`	An alias for `long`.
`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.
`date`	Date with second precision. The default format is YYYY-MM-DDTHH:MM:SS.

Datatype	Description
`decimal`	Used to represent numbers that contain decimal points. Accepts two mandatory parameters, `precision` and `scale`, which determine the range of values that can be stored in a `decimal` field. `precision` specifies the total number (from 1 to 34) of digits that can be stored. `scale` specifies the number of digits (from 0 to `precision`) that can be stored to the right of the decimal point.
`double`	An alias for `float`.
`float`	A 64-bit numeric floating point with double precision. The range of allowed values is approximately -10^{308} through $+10^{308}$.
`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.
`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.

Datatype	Description
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.
money	A legacy datatype maintained for backward compatibility. It is a signed 64-bit integer that supports 4 digits after the decimal point. Currency symbols and commas are not supported in the input data stream.

Datatype	Description
`money(n)`	A signed 64-bit numerical value that supports varying scale, from 1 to 15 digits after the decimal point. Currency symbols and commas are not supported in the input data stream, however, decimal points are. The supported range of values change, depending on the specified scale. `money(1)`: -922337203685477580.8 to 922337203685477580.7 `money(2)`: -92233720368547758.08 to 92233720368547758.07 `money(3)`: -9223372036854775.808 to 9223372036854775.807 `money(4)`: -922337203685477.5808 to 922337203685477.5807 `money(5)`: -92233720368547.75808 to 92233720368547.75807 `money(6)`: -92233720368547.75808 to 92233720368547.75807 `money(7)`: -922337203685.4775808 to 922337203685.4775807 `money(8)`: -92233720368.54775808 to 92233720368.54775807 `money(9)`: -9223372036.854775808 to 9223372036.854775807 `money(10)`: -922337203.6854775808 to 922337203.6854775807 `money(11)`: -92233720.36854775808 to 92233720.36854775807 `money(12)`: -9223372.036854775808 to 9223,372.036854775807 `money(13)`: -922337.2036854775808 to 922337.2036854775807 `money(14)`: -92233.72036854775808 to 92233.72036854775807 `money(15)`: -9223.372036854775808 to 9223.372036854775807 To initialize a variable, parameter, or column with a value of -92,233.72036854775807, specify (-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.
`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. This size limit is reduced proportionally by the size of other content in the row, including the header.

Datatype	Description
`time`	Stores the time of day as a two-byte field having a range of 00:00:00 to 23:59:59. The default format is HH24:MM:SS.
`timestamp`	Timestamp with millisecond precision. The default format is YYYY-MM-DDTHH:MM:SS:SSS.

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
<	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 pattern_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.

To specify a column, include both the column name and the stream or window name, using the format `source.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.

Literals

The terms literal and constant value are synonymous and refer to a fixed data value. For example, STOCK, SUDDEN ALERT, and 512 are all string literals; 1024 is a numeric literal.

In Event Stream Processor, string literals are delimited by single (' ') quotation marks to distinguish them from object names, which are delimited by double (" ") quotation marks.

Neither BLOB nor XML data types have literals.

Time Literals

Use time literals to specify timestamps and intervals.

Timestamp Literals

The syntax of a timestamp literal is: *`TIMESTAMP`* `'YYYY-MM-DD [HH:MI[:SS[.FF]]]'`

Where:

- YYYY-MM-DD are numeric designations of the year, month, and day.
- HH:MI are numeric designations for hour and minute.
- :SS is a designation for seconds, used only if the hour and minute are specified.
- .FF is a designation for fractions of a second, using zero to six digits and only if seconds are specified.

Use one or more blank spaces to separate the date from the time specification.

Some valid timestamps are:

```
TIMESTAMP '2002-03-12 06:05:32.474003'
TIMESTAMP '2005-02-01'
TIMESTAMP '2003-11-30 15:04'
```

In some contexts, such as when putting row timestamps into CSV files, timestamps can be entered as a number of microseconds elapsed since midnight January 1, 1970. In this case, the numbers are treated as though they are relative to UTC, rather than local time. For example, if you use 1 as the timestamp, and your local time zone is Pacific Standard Time (eight hours behind UTC), the result is the following timestamp:

```
1969-12-31 16:00:00.000001
```

Interval Literals

Use either of two formats for an interval literal. The first form is similar to that of timestamp literals:

```
INTERVAL '{[D [day[s]]][ ][HH:MI[:SS[.FF]]]}'
```

Where:

- D is the number of days. The space between the day specification and the hour and minute specification is optional.
- HH:MI are numeric designations for hour and minute.
- :SS is a designation for seconds, used only if the hour and minute are specified.
- .FF is a designation for fractions of a second, using zero to six digits and only if seconds are specified.

The following sample illustrates this syntax:

```
INTERVAL '999 days 23:59:59.999999'
```

The alternative syntax for interval literals is:

```
{[D  day[s]][ ][HH hour[s]][ ][MI  minute[s]][ ][SS[.FF] second[s]]
[ ][NNN millisecond[s][ ][NNN  microsecond[s>
```

All components of the interval are optional. Here is an example:

```
4 minutes 5.6 seconds
```

Both forms of interval literals require that the values in each component be in the proper range. For example, you will get an error if you enter 61 minutes; you must enter this value as 1 hour 1 minute.

Boolean Literals

Boolean literals are True and False statements which are not sensitive to case. For example, True, false, TRUE, FALSE, tRuE, fAlSe, true, False, truE, and falsE are all valid.

Values other than true or false - such as Y/N or 1/0 - are invalid.

String Literals

String literals appear as a part of expressions.

String literals are also sometimes called character literals or text literals. When a string literal appears as part of an expression in this documentation, it is indicated by the word TEXT. The syntax for for both single-line and multi-line string literals is:

```
'character_string'
```

In all cases, `character_string` is a combination of alphabetic characters, numeric characters, punctuation marks, spaces, and tabs. In addition:

- Both single-line and multi-line string literals must be enclosed in single (' ') quotation marks.
- Double (" ") quotation marks can be used as part of a string.

- Two adjacent single quotation marks with no character string between represent an empty string.

Note: Double quotation marks used to delimit a string are used for object names and not string literals. Object names cannot be multiple lines long or contain newline (NL) characters or double quotation marks within the string.

To include a single quotation mark (or an apostrophe) in a string delimited by single quotation marks, enter a backslash before the quotation mark (\') for the inside quotation marks. For example:

```
'And that\'s the truth.'
```

To include a single quotation mark (or an apostrophe) in a string literal (delimited by single quotation marks), enter a backslash before the quotation mark or apostrophe you want to include in the string: (\'). For example:

```
'He said "No!"'
```

Some examples of valid string literals are:

```
'abc123'
'abc 123'
'It\'s a good idea.'
'"What?" he asked.'
```

Internationalization impacts string literals. All the literals in the preceding list are 7-bit ASCII literals. But this is also a literal:

```
'123'
```

Numeric Literals

Numeric literals are used to specify integers, long, and floating-point numbers.

Integer Literals

Use the integer notation to specify integers in expressions, conditions, functions, and CCL statements.

The syntax of an integer literal is:

```
[+|-]integer
```

where `integer` refers to any whole numbers (including negatives) or zero.

Some valid integers are:

```
3
-45
+10023
```

Long Literals

Long literals follow the same rules as integer literals. To force a literal that can be either integer or long into a long data type, add the letter "L" to the end of the literal.

For example, the following are valid long literals:

```
2147483648L
-2147483649L
-9223372036854775808L
0L
```

Float Literals

A float literal is a floating-point number, usually used to represent numbers that include a decimal point. Use the float literal syntax whenever an expression requires a floating point number.

The syntax of a float literal is:

```
[+|-]floating_point_number
[E[+|-]exponent]
```

where `floating_point_number` is a number that includes a decimal point. The optional letter e or E indicates that the number is specified in scientific notation. The digits after the E specify the exponent. The exponent can range from approximately -308 to +308.

Some valid float literals are:

```
1.234
-45.02
+10023.
3.
.024
-7.2e+22
```

Note that float values are accurate to 16 significant digits.

CHAPTER 5 **CCL Query Construction**

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

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

Filtering

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

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

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

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

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

Splitting Up Incoming Data

Use the SPLITTER construct to separate incoming data according to filtering rules and write it out to different target streams.

When you want to separate incoming data into several subsets and process those subsets differently, use the **CREATE SPLITTER** construct, which operates like the ANSI **case** statement. It reads the incoming data, applies the specified filtering conditions and writes out each subset of the data to one or more target streams.

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.

The first condition that evaluates to true (non-zero value) causes the record as projected in the column_list to be inserted into the corresponding target streams. Subsequent conditions are neither considered nor evaluated. 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.

Example

The example creates a schema named TradeSchema and applies that schema to the input window Trades. IBM_MSFT_Splitter evaluates and routes data to one of three output windows. Event records with the symbol IBM or MSFT are sent to the IBM_MSFT_Tradeswin window. Event records where the product of `trw.Price * trw.Volume` is greater than 25,000 are sent to the Large_TradesWin window. All event records that do not meet the conditions placed on the two previous output windows are sent to the Other_Trades window.

```
CREATE SCHEMA TradeSchema (
Id long,
Symbol STRING,
Price MONEY(4),
Volume INTEGER,
TradeTime DATE
) ;
CREATE INPUT WINDOW Trades
SCHEMA TradeSchema
PRIMARY KEY (Id) ;
CREATE SPLITTER IBM_MSFT_Splitter
AS
WHEN trw.Symbol IN ('IBM', 'MSFT') THEN IBM_MSFT_Trades
WHEN trw.Price * trw.Volume > 25000 THEN Large_Trades
ELSE Other_Trades
SELECT trw. * FROM Trades trw ;
CREATE OUTPUT WINDOW IBM_MSFT_TradesWin
PRIMARY KEY DEDUCED
AS SELECT * FROM IBM_MSFT_Trades ;
CREATE OUTPUT WINDOW Large_TradesWin
PRIMARY KEY DEDUCED
AS SELECT * FROM Large_Trades ;
CREATE OUTPUT WINDOW Other_TradesWin
```

```
PRIMARY KEY DEDUCED
AS SELECT * FROM Other_Trades ;
```

Unions

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

If the UNION is on a Window or Delta Stream, duplicate rows are eliminated from the result set due to the primary key. If the UNION is on a Stream, duplicates flow through.

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

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

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

Example: Merging Data from Streams or Windows

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

1. Create a new window:

   ```
   CREATE WINDOW name
   ```

   You can also create a new stream or delta stream.
2. Specify the primary key:

   ```
   PRIMARY KEY (…)
   ```
3. Specify the first derived element in the union:

```
SELECT * FROM StreamWindow1
```

4. Add the **UNION** clause:

```
UNION
```

5. Specify the second derived element in the union:

```
SELECT * FROM StreamWindow2
```

Joins

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

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

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

Note: Unnamed windows are implicitly created when using a join with a window that produces a stream. The unnamed window is created to ensure that a join does not see records that have not yet arrived at the join. This can happen because the source to the join and the join itself are running in separate threads.

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

Joins in ANSI syntax can add the **DYNAMIC** modifier to a window or stream to indicate that its data changes frequently. 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.

Event Stream Processor supports all join types:

Join Type	Syntax	Description
Inner Join	`INNER JOIN`	One record from each side of the join is required for the join to produce a record.

Join Type	Syntax	Description
Left Outer Join	`LEFT JOIN`	A record from the left side (outer side) of the join is produced regardless of whether a record exists on the right side (inner side). When a record on the right side does not exist, any column from the inner side has a NULL value.
Right Outer Join	`RIGHT JOIN`	Reverse of left outer join, where the right side is the outer side and the left side is the inner side of the join.
Full Outer Join	`FULL JOIN`	A record is produced whether there is a match on the right side or the left side of the join.

Event Stream Processor also supports these cardinalities:

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

Note: When a join produces multiple rows, the rows are grouped into a single transaction. If the transaction fails, all of the rows are discarded.

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

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

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

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

Key Field Rules

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

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

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

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

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

	One-One	One-Many	Many-One	Many-Many
OUTER	Keys should be formed using **firstnonnull** () on each pair of keys from both sides.	Not allowed.	Not allowed.	Not allowed.

See also

- *Join Examples: ANSI Syntax* on page 49
- *Join Example: Comma-Separated Syntax* on page 51

Join Examples: ANSI Syntax

Examples of different join types using the ANSI syntax.

Refer to these inputs for the examples below.

```
CREATE INPUT STREAM S1 SCHEMA (Val1S1 integer, Val2S1 integer, Val3S1
string);
CREATE INPUT WINDOW W1 SCHEMA (Key1W1 integer, Key2W1 string, Val1W1
integer, Val2W1 string) PRIMARY KEY (Key1W1, Key2W1);
CREATE INPUT WINDOW W2 SCHEMA (Key1W2 integer,  Key2W2 string, Val1W2
integer, Val2W2 string) PRIMARY KEY (Key1W2, Key2W2);
CREATE INPUT WINDOW W3 SCHEMA (Key1W3 integer,  Val1W3 integer,
Val2W3 string) PRIMARY KEY (Key1W3);
```

Simple Inner Join: One-One

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

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

Simple Left Join: One-One

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

```
CREATE OUTPUT WINDOW OW2
PRIMARY KEY (Key1W1, Key2W1)
AS SELECT W1.*, W2.*
FROM W1 LEFT JOIN W2 ON W1.Key1W1 = W2.Key1W2 AND W1.Key2W1 =
W2.Key2W2;
```

Simple Full Outer Join: One-One

The key columns all have a required **firstnonnull** expression in it.

```
CREATE OUTPUT WINDOW OW3
PRIMARY KEY (Key1, Key2)
AS SELECT firstnonnull(W1.Key1W1, W2.Key1W2) Key1,
firstnonnull(W1.Key2W1, W2.Key2W2) Key2, W1.*, W2.*
FROM W1 FULL JOIN W2 ON W1.Key1W1 = W2.Key1W2 AND W1.Key2W1 =
W2.Key2W2;
```

Simple Left Join: Many-One

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

```
CREATE OUTPUT WINDOW OW4
PRIMARY KEY (Key1W1, Key2W1)
AS SELECT W1.*, W2.*
FROM W1 LEFT JOIN W2 ON W1.Key1W1 = W2.Key1W2 AND W1.Val2W1 =
W2.Key2W2;
```

Simple Left Join: Many-One (DYNAMIC Modifier)

W3 is **DYNAMIC** and only one key of W1 is mapped in this join, so a secondary index is created on W1. W1 is also **DYNAMIC**, but all keys of W3 are mapped, so no secondary index is created on W3.

```
CREATE OUTPUT WINDOW OW5
PRIMARY KEY DEDUCED
AS SELECT W1.*, W3.*
FROM W1 (DYNAMIC) LEFT JOIN W3 (DYNAMIC) ON W1.Key1W1 = W3.Key1W3;
```

Simple Inner Join: Many-Many

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

```
CREATE OUTPUT WINDOW OW6
PRIMARY KEY (Key1W1, Key2W1, Key1W2, Key2W2)
AS SELECT W1.*, W2.*
FROM W1 JOIN W2 ON W1.Val1W1 = W2.Val1W2 AND W1.Val2W1 = W2.Val2W2;
```

Simple Stream-Window Left Join

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

```
CREATE OUTPUT STREAM OSW1
AS SELECT S1.*, W2.*
FROM S1 LEFT JOIN W2 ON S1.Val1S1 = W2.Key1W2 AND S1.Val3S1 =
W2.Key2W2;
```

Complex Window-Window Join

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

```
CREATE OUTPUT WINDOW OW7
PRIMARY KEY DEDUCED
AS SELECT S1.*, W1.*, W2.*, W3.*    //Some column expression.
FROM S1 LEFT JOIN (W1 INNER JOIN (W2 LEFT JOIN W3 ON W2.Key1W2 =
W3.Key1W3) ON W1.Key1W1 = W2.Key1W2 AND W1.Key2W1 = W2.Key2W2) ON
S1.Val1S1 = W1.Key1W1
```

```
WHERE W2.Key2W2 = 'abcd'
GROUP BY W1.Key1W1, W2.Key2W2
HAVING SUM(W3.Val1W3) > 10;
```

Complex Stream-Window Join

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

```
CREATE OUTPUT WINDOW OW8
PRIMARY KEY DEDUCED
AS SELECT S1.*, W1.*, W2.*, W3.*   //Some column expression.
FROM S1 LEFT JOIN (W1 INNER JOIN (W2 LEFT JOIN W3 ON W2.Key1W2 =
W3.Key1W3) ON W1.Key1W1 = W2.Key1W2 AND W1.Key2W1 = W2.Key2W2) ON
S1.Val1S1 = W1.Key1W1
WHERE W2.Key2W2 = 'abcd'
GROUP BY W1.Key1W1, W2.Key2W2
HAVING SUM(W3.Val1W3) > 10;
```

See also

- *Key Field Rules* on page 48
- *Join Example: Comma-Separated Syntax* on page 51

Join Example: Comma-Separated Syntax

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

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

```
CREATE OUTPUT WINDOW OW4
PRIMARY KEY DEDUCED
AS SELECT W1.*, W2.*, W3.*
FROM W1, W2, W3
WHERE W1.Key1W1 = W2.Key1W2 AND W1.Key2W1 = W2.Key2W2  AND W1.Key1W1
= W3.Key1W3;
```

See also

- *Key Field Rules* on page 48
- *Join Examples: ANSI Syntax* on page 49

Pattern Matching

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

Pattern streams can check whether or not events (rows from the input sources matching certain criteria) occur during a specific time interval, and then send records to downstream streams if a match has occurred.

Pattern matching can be used to distill complex relationships between data into compact and easily-maintainable expressions.

Attention: The pattern rule engine will use any incoming event in order to match the defined pattern, regardless of the opcode of an incoming event. The opcode can be included in each event's definition in order to filter out unwanted rows.

This example creates an output stream, `ThreeTrades`, which monitors the QTrades streams and sends a new event when it detects three trades on the same symbol within five seconds. The output of this stream is the symbol of the traded stock, and its latest three prices. The trades do not have to occur consecutively, but the trades must occur within five seconds of each other. Multiple patterns may be in the process of being matched at the same time.

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

For details on the **MATCHING** clause, see the *SAP Sybase Event Stream Processor Programmers Reference.*

Aggregation

Aggregation collects input records based on the values in the columns specified with the **GROUP BY** clause, applies the specified aggregation function such as `min`, `max`, `sum`, `count` and so forth, and produces one row of output per group.

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

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

The **GROUP FILTER** clause is executed after the **GROUP ORDER BY** clause and eliminates any rows in the group that do not meet the filter condition. The filter condition that is specified is similar to the one in the **WHERE** clause. The only exception being that a special rank function can be specified. The rank function is used in conjunction with the **GROUP ORDER BY** clause. After the **GROUP ORDER BY** clause is executed every row in the group is ranked from 1 to N. Now in the **GROUP FILTER** clause one can say `rank() < 11`, which means that the aggregation function is only applied to the first 10 rows in the group after it has been ordered by the columns specified in the **GROUP ORDER BY** clause.

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

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

When using aggregation, you must consider the memory usage implications. All of the input records for which an aggregate function is to be calculated have to be stored in memory. The data structure that holds all the records in memory is called the aggregation index.

If a stream feeds input to an aggregation window directly, the memory usage of the aggregation index increases without bound. To prevent such unbounded growth, insert an intermediate window between the stream and the aggregation window. In this intermediate window, use a **GROUP BY** clause to set one or more of the stream columns as the primary key, then set a retention policy to prevent runaway memory usage. Note that although this intermediate window is an aggregation window, it does not perform any aggregation functions, so its index does not grow indefinitely.

The intermediate aggregation window acts as the stream and feeds input into the aggregation window directly. The aggregation window performs its aggregation functions using the input retained from the intermediate aggregation window.

Example

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

```
CREATE INPUT STREAM Trades
SCHEMA (TradeId integer, Symbol string, Price float, Shares integer);

CREATE WINDOW TradeRetention
PRIMARY KEY DEDUCED
KEEP 1 DAY
AS
SELECT trd.TradeID, trd.Symbol, trd.Price, trd.Shares
FROM Trades trd
GROUP BY trd.TradeId;

CREATE OUTPUT WINDOW TradeSummary
PRIMARY KEY DEDUCED
KEEP 1 DAY
AS
```

```
SELECT tr.Symbol, count (tr.TradeId) NoOfTrades, max (tr.Price)
MaxPrice, sum(tr.Shares) TotalShares
FROM TradeRetention tr
GROUP BY tr.Symbol
HAVING sum(tr.Shares) > 5000;
```

Reference Table Queries

Reference table queries enable you to look up information in an external database table in response to an incoming event.

Incoming events can arrive with only a portion of the information necessary to complete the processing you wish to specify in your project. When that additional information is present in existing tables in an external database, you can use reference table queries to look it up. There are two parts to this: creating the reference to an external table and using the reference to execute an ad hoc query in a join or flex operator.

When creating the reference, you must first decide what data you want to use. Then identify the external database table containing the data by name, obtain the schema of the table, and find out what service to use to contact the database. Decide whether you want to attempt to reconnect if the connection is dropped, and if so, how many times, and how long to wait between attempts.

When joining a stream or window to a reference, you need to decide what you want as the output of the join. There are numerous restrictions on how you can use references in joins, and what types of output you can obtain. For example, if you want the output of the join to be a window, you must specify the primary key of the reference and use the complete primary key in the **ON** or **WHERE** clause of the join.

There are several different ways to use references within SPLASH programs. You can iterate over the rows in the table or grab specific rows. Basically, you can utilize a reference in the same ways you can utilize a window. It is simply another source of data for processing in your SPLASH routine.

You can use references - in joins and in SPLASH programs - inside a module as well as within the main body of your project. Like stores, references used in a module must be bound to a reference defined in the main body of your project.

Prerequisites

You must have the SAP HANA ODBC client installed on the system where you want to run projects that include reference table queries. SAP recommends that you use the latest version of the SAP HANA ODBC client available, but it must be at least version 1.0.67.

Database Service Definition

All connections to external databases, including reference table queries, are made using services defined in the `services.xml` file. This file, as shipped with ESP, includes a service definition, **SampleHanaJdbc** , that shows how to define a service for use in reference table queries to a HANA database table. On Windows machines it is in the *%ESP_HOME%*`\bin` folder. On UNIX and Linux machines it is in the *$ESPHOME*`/bin` directory. You can modify entries in this file, or add new ones, for use in reference table queries.

Error Handling

When you start a project that contains a reference table query, it does a table schema compatibility check. The reference scans the database table and verifies that:

- For each column specified in the reference, there is a column of the same name (case insensitive) in the table.
- The datatype of the column in the table is compatible with the datatype of the column in the reference.
- If the reference definition specifies a primary key, there is a matching primary key in the database. (If the reference definition doesn't specify a primary key, it doesn't matter whether or not the database has a primary key.)

In order to check the type for each mapped column, the reference attempts to pull a sample row from the database. It's done this way to be as database-agnostic as possible. If it can pull that column into ESP, the check succeeds. Otherwise it fails, except in the following two cases:

- If the query that the reference uses to do the type-checking is rejected by the database (because it doesn't support SQL 2003 standards), the reference will not complete type checking, but will allow the project to start up, providing a warning that it can't guarantee that the type mapping is valid.
- If the table has no data in it, then the type checking will stop, and a warning will be printed that it can't guarantee that the type mapping is valid.

While a project is running, the error scenarios are mostly connection-based. When a failure is caused by a lost connection, the server will attempt to reconnect based on the reconnect parameters specified in the reference's definition.

Using a Reference Table Query in a Join

This example shows the procedure for creating a reference table query, an input stream, and an output stream that implements a join of data from the reference table query and the input stream to add customer data to an incoming order.

1. Obtain the necessary information about the external database table containing the information you wish to look up.
   - the name of the table containing the data
   - the schema for that table
   - the service to use to connect to it
2. Create the reference table query in your project.

For example, for a table named **customerAddr**, with an integer column, **customerID**, a string column, **fullName**, and a string column, **address**, that can be accessed by a service named **databaseLookup**, create a reference table query named **customerRef**:

```
CREATE REFERENCE customerRef
SCHEMA (customerID integer, fullName string, address string)
PRIMARY KEY (customerID)
PROPERTIES
service='databaseLookup',source='customerAddr',sourceSchema='addr
essSchema';
```

The schema of the reference does not have to include all of the columns in the source table or list them in the same order. But, for each column that you do specify in the schema, the column name must match the column name in the database, and the datatype must be compatible with the datatype in the database.

The **customerID** column is the primary key in the source table: it contains a unique number assigned to each customer. A primary key is not required to produce a stream: only a window. But, if you declare a primary key for the reference it must match the primary key in the table. So, whenever the table has a primary key, it is a good practice to make it the primary key of the reference table query (giving you the flexibility to produce either a stream or a window).

When you specify a primary key in a reference table query, you must also include the **sourceSchema** property, which identifies the schema, in this case **addressSchema**, of the database containing the table.

3. Create the input stream for events to enter the project.

   For an event which is an order placed by a customer, consisting of three integers, named **orderID**, **customerID**, and **itemID**, create an input stream named **orderStream**.

   ```
   CREATE INPUT STREAM orderStream
   SCHEMA (orderID integer, customerID integer, itemID integer);
   ```

4. Create an output stream that implements a join of the incoming event with the data obtained from the reference table query.

   To create an output stream named **orderWithCustomerInfoStream** with three columns: the integer **orderID**, the string **customerName**, and the string **customerAddress**, the output stream joins the **orderID** from the **orderStream** stream, with the **fullname** and **address** from the **customerRef** reference table query when the **customerID** in the **orderStream** stream matches the **customerID** in the **customerRef** reference table query.

   ```
   CREATE OUTPUT STREAM orderWithCustomerInfoStream
   SCHEMA (orderID integer, customerName string, customerAddress
   string)
   AS SELECT orderStream.orderID, customerRef.fullName,
   customerRef.address
   FROM orderStream, customerRef
   WHERE orderStream.customerID=customerRef.customerID;
   ```

While the column names in the output stream's schema declaration, **customerName** and **customerAddress**, do not have to match the column names in the table, **fullName** and **address**, the datatypes in the output stream's schema must match the datatypes in the table.

Using a Reference Table Query in a Module

This example obtains a bibliography for the specified author using a Reference Table Query inside of a module.

In this example, an order for a book produces a list of other books by that author. Because this list may be desired in response to other events, it is produced within a module.

1. Obtain the necessary information about the external database table containing the information you wish to look up.
   - the name of the table containing the data
   - the schema for that table
   - the service to use to connect to it
2. Create the reference table query in your project. When you define a reference in a module, it must always be mapped to a reference in your project which contains the connection information to the database.

   For example, for a table named **bibliographies**, with a string column, **authorName**, a string column, **bookTitle**, and a string column, **publisher**, that can be accessed by a service named **databaseLookup**, create a reference table query named **getBiblio**:

   ```
   CREATE REFERENCE getBiblio
   SCHEMA (authorName string, bookTitle string, publisher string)
   PROPERTIES
   service='databaseLookup',source='bibliographies',sourceSchema='bo
   okSchema';
   ```
3. Create an input stream for order events to enter the project.

   For an event which is an order placed by a customer, consisting of three integers, named **orderID**, **customerID**, and **itemID**, and one string **authorName** create an input stream named **orderStream**.

   ```
   CREATE INPUT STREAM orderStream
   SCHEMA (orderID integer, customerID integer, itemID integer,
   authorName string)
   ;
   ```
4. Create an output stream named **otherBooks** with three string columns: **authorName**, **bookTitle**, and **publisher** to display the list of other books by the author.

   ```
   CREATE OUTPUT STREAM otherBooks
   SCHEMA (authorName string, bookTitle string, publisher string)
   ;
   ```
5. Create a module that uses this table reference query. Note that in the module you only need to specify the name and schema of the reference table query.

   ```
   CREATE MODULE listOtherBooks
   IN orderStream
   ```

```
OUT otherBooks
REFERENCES getBiblio
BEGIN
    CREATE REFERENCE getBiblio
        SCHEMA (authorName string, bookTitle string, publisher
string);
    CREATE INPUT STREAM orderStream
        SCHEMA (orderID integer, customerID integer, itemID
integer, authorName string);
    CREATE OUTPUT STREAM otherBooks
        SCHEMA (authorName string, bookTitle string, publisher
string)
        FROM orderStream, bibliography
        WHERE orderStream.authorName=bibliography.authorName;
END;
```

6. Load the module to generate the list of other works by the author of the book that was ordered. Loading the module creates a mapping between the streams in the main project and in the module. The output from the module will be visible in the **otherBooks** stream in the main project.

```
LOAD MODULE listOtherBooks AS listAll
    IN
        orderStream = orderStream
    OUT
        otherBooks = otherBooks
    REFERENCES
        getBiblio = getBiblio
;
```

Using a Reference Table Query in SPLASH

This example uses a reference table query within a SPLASH routine that iterates over the rows in a table to obtain data about a specified player's batting history that is then used to compute statistics.

1. Obtain the necessary information about the external database table containing the information you wish to look up.
   - the name of the table containing the data
   - the schema for that table
   - the service to use to connect to it
2. Create the reference table query in your project.

   For example, for a table named **Batting**, with an integer column, **teamAtBat**, a string column, **player**, and an integer column, **bases**, that can be accessed by a service named **HANA**, create a reference table query named **AtBats**:

```
CREATE REFERENCE AtBats
SCHEMA (teamAtBat INTEGER, player STRING, bases INTEGER)
PRIMARY KEY (teamAtBat)
PROPERTIES
service='HANA',source='Batting',sourceSchema='BattingSchema';
```

The **teamAtBat** is incremented each time a player bats, providing a unique number for each at bat by a member of the team, so that column is the primary key in the source table. It must also be the primary key of the reference table query. Because we have specified a primary key in this reference, we have to include the **sourceSchema** property, which identifies the schema, **BattingSchema**, of the database containing the table.

3. Create the input stream for events to enter the project.

For an event which is the appearance, or mention, of a player's name, create an input stream named **StatRequest**, consisting of a single string, **player**.

```
CREATE INPUT STREAM StatRequest SCHEMA (player STRING);
```

4. Initialize the Flex operator, named **Percentage**, to read in the reference query **AtBats** and the input stream **StatRequest**, specify the output stream **PercentageOutput**, and set the variables *atBats* and *hits* to zero.

```
CREATE FLEX Percentage IN AtBats, StatRequest
OUT OUTPUT STREAM PercentageOutput SCHEMA (atBatPercentage float)
BEGIN
   ON StatRequest {
      integer atBats := 0;
      integer hits := 0;

```

5. Close the Flex operator by dividing the number of *hits* by the number of *atBats* to calculate the batting average, *percentage*, placing that value in the *atBatPercentage* column of the output stream, and outputting it.

```
      }
      float percentage := hits/atBats;
      output [atBatPercentage=percentage];
   };
END;
```

6. Between the initialization and close of the Flex operator, add a SPLASH routine to iterate through the entries in the **Batting** table to find out how many times the player was at bat and how many hits he got. There are three ways to do this.

a) Use the **for** and **if** commands to handle iterating through the records, and take advantage of the *AtBats_stream* that is automatically created with the **AtBats** reference query.

```
      for ( record in AtBats_stream ) {
         if ( record.player = StatRequest.player ) {
            if ( record.bases > 0 ) {
               hits := hits + 1;
            }
            atBats := atBats + 1;
         }
      }
```

This method has the advantage of being easy to read and maintain.

b) Use the modified **for** command syntax that incorporates testing whether the player name from the input stream matches the player name in the table entry.

```
      for ( record in AtBats_stream where player =
StatRequest.player ) {
         if ( record.bases > 0 ) {
            hits:= hits + 1;
         }
         atBats := atBats + 1;
      }
```

This method has the advantage of reducing processing time, because the records of other players are never pulled from the reference table.

c) Explicitly create and modify the iterator to use across the table, again taking advantage of the *AtBats_stream* that is automatically created with the **AtBats** reference query.

```
     AtBats_iterator := getIterator( AtBats_stream );
    resetIterator( AtBats_iterator);
    setRange( AtBats_iterator, player, StatRequest.player );
    typeof(AtBats) result := getNext( AtBats_iterator );
    while (not(isnull(result))) {
      if ( result.bases > 0 ) {
        hits := hits + 1;
      }
      atBats := atBats + 1;
      result := getNext( AtBats_iterator );
    }
```

This method has the advantage of giving you the most explicit control over the processing.

Using a Reference Table Query in SPLASH to Get a Single Row from a Table

This example uses a reference table query within a SPLASH routine to obtain a single row from a table.

In this example, we want to obtain a list of chain restaurants that have a franchise in a specified city.

1. Obtain the necessary information about the external database table containing the row you want to obtain.
   - the name of the table containing the data
   - the schema for that table
   - the service to use to connect to it
2. Create a second reference table query in your project.

   For example, for a table named **chainList**, with an integer column, **chainID**, a string column, **city**, a string column, **name**, and a string column, **headquartersAddress**, that can be accessed by a service named **HANA**, create a reference table query named **chainList**:

   ```
   CREATE REFERENCE restaurants
   ```

```
SCHEMA (storeID INTEGER, city STRING, name STRING, address STRING)
PROPERTIES service='HANA',source='restaurants';
```

Again, because we have not specified a primary key in this reference, we can omit the **sourceSchema** property.

3. Create the input stream for events to enter the project.

   Create an input stream named **restaurantQuery**, consisting of a single string, **city**.

   ```
   CREATE INPUT STREAM StatRequest SCHEMA (player STRING);
   ```

4. Initialize the Flex operator, named **optionQuery**, to read in the reference table queries **restaurants** and **chainList**, and the input stream **restaurantQuery**, output the restaurant chain names in the output stream **chainsInCity** when it receives the name of a city in the **restaurantQuery** stream.

   ```
   CREATE FLEX optionQuery IN restaurants, chainList, restaurantQuery
   OUT OUTPUT STREAM chainsInCity SCHEMA (name string, address
   string)
   BEGIN
      ON restaurantQuery {
   ```

5. Add a SPLASH routine to produce the list of chains that have a presence in the city. For each restaurant chain in the **chainList** table, it grabs the first instance of that chain name in the **restaurants** table whose address is in the city specified in the **restaurantQuery** input stream and outputs the restaurant chain name and address.

   ```
         for (record in chainList_stream) {
            typeof(restaurants) inCity :=
   restaurants_stream{[city=restaturantQuery.city;name=record.name;]
   };
            output [name=inCity.name;address=restaurants.address;];
         }
   ```

6. Since we are not performing any calculations this time, simply close the Flex operator.

   ```
      };
   END;
   ```

CHAPTER 6 Advanced CCL Programming Techniques

Use advanced CCL techniques to develop sophisticated and complex projects.

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

Create modules to encapsulate reusable code.

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

Declare Blocks

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

CCL supports global and local declare blocks.

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

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

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

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

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

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

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

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

Typedefs

Declares new names for existing datatypes.

Syntax

```
typedef existingdatatypeName newdatatypeName;
```

Components

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

Usage

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

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

Example

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

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

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

```
euros price := 10.80d2;
```

which is the same as:

```
money(2) price := 10.80d2;
```

Parameters

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

Syntax

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

Components

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

Usage

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

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

Parameters at Project Setup

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

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

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

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

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

Variables

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

Syntax

```
typeName {variableName[:=any_expression] [, ...]}
```

Usage

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

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

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

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

Example

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

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

Create Schema ControlSchema (
    Msg STRING,
    Value   INTEGER
); //
// Input  Trade Window
//

CREATE  INPUT  WINDOW TradeWindow
  SCHEMA TradeSchema
  PRIMARY KEY (Ts);

//
// Input Stream for Control Messages
//

CREATE INPUT STREAM ControlMsg SCHEMA ControlSchema ;

//
// Output window, only has rows that were greater than the
thresholdvalue
// was when the row was received
CREATE  Output  WINDOW OutTradeWindow
    SCHEMA (Ts bigdatetime, Symbol STRING, Price MONEY(4), Volume
INTEGER)
    PRIMARY KEY (Ts)
as
select *
    from TradeWindow
    where TradeWindow.Volume > ThresholdValue;

//
//Flex Stream to process the control message
CREATE FLEX FlexControlStream
  IN ControlMsg
  OUT OUTPUT WINDOW SimpleOutput
  SCHEMA ( a integer, b string, c integer)
     PRIMARY KEY ( a)
BEGIN
    ON ControlMsg
    {
        // change the value of ThresholdValue
        if ( ControlMsg.Msg = 'set')
{ThresholdValue:=ControlMsg.Value;}
        // The following is being populate so you can see that the
ThresholdValue is being set
        output [a=ControlMsg.Value; b=ControlMsg.Msg;
c=ThresholdValue; |];
    }
    ;
END
;
```

Declaring Project Variables, Parameters, Datatypes, and Functions

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

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

   Elements defined in this declare block are accessible to any elements in the project that are not inside a module.
3. Create local declare blocks by using the **DECLARE** statement within derived streams, windows, or both..
4. Add variables, parameters, or user-defined SPLASH functions to the local declare block.

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

Flex Operators

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

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

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

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

Data structures	• Variables • EventCache (windows) • Dictionaries • Vectors
Control structures	• While • If • For

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

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

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.

Automatic Partitioning

You can improve the performance of a given element and complex projects, which perform computationally expensive operations such as aggregation and joins, by using automatic partitioning. Automatic partitioning is the creation of parallel instances of a given element and partitioning input data into these instances. Partitioning data this way results in higher performance as the workload is split across the parallel instances.

Usage

You can create parallel instances of a delta stream, stream, window, or module by using the PARTITION BY clause. Add this clause within a CCL statement such as CREATE DELTA STREAM, CREATE STREAM, CREATE WINDOW, or LOAD MODULE. Specify the partitioning degree, elements to be partitioned, and a partitioning function.

The partitioning degree is the natural number of parallel instances you wish to create for a given element (delta stream, stream, window, or module). As an alternative to specifying the partitioning degree as a constant, you can specify it using an integer parameter with an optional default value. You can then provide the actual value for the parameter in the CCR project configuration file.

When using the PARTITION BY clause, partition at least one input stream of the corresponding element by specifying a partitioning function. In the case that an element accepts multiple input streams and some of these input streams do not have a partitioning function defined, those streams are broadcast to all parallel instances.

The partitioning function is effectively a demultiplexer which determines the target parallel instances for a given partitioning key. There are three valid types of partition functions: ROUNDROBIN, HASH, and CUSTOM. Choose a type based on the calculations you are performing on the input data. For example, ROUNDROBIN is sufficient for stateless operations like simple filters, but not for aggregation as this would produce differing results. HASH is necessary for grouping records together, but grouping may not evenly distribute the data across instances.

The CUSTOM partitioning function is defined as an inline function which does not take any parameters. This function creates an implicit global parameter called **<targetName>_partitions** where **<targetName>** represents the name of the current element you are partitioning and **partitions** is a fixed part of the parameter name. For example, if you are partitioning an output window called `maxPriceW`, use **maxPriceW_partitions** as the global parameter name. The value of this parameter is equal to the number of partitions.

```
create input stream priceW schema (isin string, price money(2));

create output window maxPriceW schema (isin string, maxPrice
money(2))
primary key deduced
keep 5 minutes
PARTITION by priceW
{
     integer hashValue := ascii(substr(priceW.isin,1,1));
     return hashValue % maxPriceW_partitions;
}
PARTITIONS 2
as
SELECT upper(left(priceW.isin,1)) isin, max(priceW.price) maxPrice
FROM priceW group by upper(left(priceW.isin,1));
```

Do not explicitly provide a runtime value for this parameter. To ensure uniqueness, the compiler throws an error if you create a global variable by the same name.

The CUSTOM partitioning function returns an integer which determines the parallel instance that should receive a given event (row). A modulo operation applies to this result, which ensures that the returned instance number is greater than or equal to zero and is less than the number of available instances. This prevents runtime errors. For example, if you create three partitions, those partitions will have the IDs 0, 1, and 2.

Ordering of Partitioned Results

Note that for the same input data the output of partitioned elements may differ from the output of a non-partitioned element. This is caused by the fact that:

- operating systems schedule threads in a non-deterministic way, and
- parallel execution of instances using multiple operating system threads introduces indeterminism, and
- to maximize the throughput of the partitioned element, no explicit synchronization between parallel instances takes place

The stream partitions which are instantiated by the ESP Server at runtime are local and cannot be subscribed or published to. However, these streams are visible in Studio so you can view their utilization and adjust the partition count accordingly.

Restrictions

You cannot apply the PARTITION BY clause to these elements: inputs, splitters, unions, reference streams, and adapters. Doing so results in a syntax error. However, you can partition these elements within a module that you are partitioning.

Example: Roundrobin Partitioning

Here is an example of ROUNDROBIN partitioning on a CCL query with one input window (`TradeWindow`):

```
create input window TradeWindow
schema (
        Ts BIGDATETIME,
        Symbol STRING,
        Price MONEY(2),
        Volume INTEGER)
primary key (Ts);

create output window TradeOutWindow
schema (
        Ts BIGDATETIME,
        Symbol STRING,
        Price MONEY(2),
        Volume INTEGER)
primary key (Ts)
PARTITION
    by TradeWindow ROUNDROBIN
PARTITIONS 2
```

```
as
SELECT * FROM TradeWindow
WHERE TradeWindow.Volume > 10000;
```

This example partitions the output window, `TradeOutWindow`, using ROUNDROBIN partitioning and creates two parallel instances.

Example: HASH Partitioning

Here is an example of HASH partitioning on a CCL query with one input window (`priceW`):

```
create input stream priceW
schema (isin string, price money(2));

create output window maxPriceW
schema (isin string, maxPrice money(2))
primary key deduced keep 5 minutes
PARTITION
    by priceW HASH(isin)
PARTITIONS 5
as
SELECT upper(left(priceW.isin,1)) isin, max(priceW.price) maxPrice
FROM priceW group by upper(left(priceW.isin,1));
```

This example partitions the output window, `maxPriceW`, using HASH partitioning and creates five parallel instances.

Here is an example of HASH partitioning on one of the input windows (`priceW`) on a join while the other input window (`volumeW`) is broadcast:

```
create input window priceW
schema (isin string, price float) primary key (isin) keep 5 minutes;

create input window volumeW
schema (isin string, volume integer)
primary key (isin) keep 5 minutes;

create output window vwapW
primary key deduced keep 1 minute
PARTITION
    by priceW HASH (isin)
PARTITIONS 2
as
SELECT priceW.isin, vwap(priceW.price, volumeW.volume) vwap_val
FROM priceW LEFT JOIN volumeW ON priceW.isin = volumeW.isin
group by priceW.isin;
```

This example partitions the output window, `vwapW`, using HASH partitioning and creates two parallel instances.

Example: CUSTOM Partitioning

Here is an example of CUSTOM partitioning on a CCL query with two input windows (`priceW` and `volumeW`):

```
create input window priceW
schema (isin string, price float)
primary key (isin) keep 5 minutes;

create input window volumeW
schema (isin string, volume integer)
primary key (isin) keep 5 minutes;


create output window vwapW
schema (isin string, vwap float)
primary key deduced
partition
by priceW {
return ascii(substr(priceW.isin,1,1)) % vwapW_partitions;
},
by volumeW {
return ascii(substr(volumeW.isin,1,1)) % vwapW_partitions;
}
partitions 2
as
SELECT priceW.isin, vwap(priceW.price, volumeW.volume) vwap_val
FROM priceW LEFT JOIN volumeW ON priceW.isin = volumeW.isin
group by priceW.isin;
```

This example partitions the output window, `vwapW`, using a CUSTOM partitioning function and creates two parallel instances.

Guidelines for Partitioning Flex Operators

General guidelines, tips, and examples of partitioning elements using flex operators.

Local Variables

Each partition has its own locally declared variables. Therefore, a counter of all input records only counts records that arrive at a certain partition. If you wish to use a locally declared counter, it is recommended that you use it in line with the given partition key and that you verify the results of the partitioned element.

Globally Declared Variables

Globally declared variables can also be tricky. Multiple partitions in parallel can change a globally declared variable in parallel and race-conditions cannot be avoided. For example, if a window has a user-defined method which increases a global counter variable by 1 at the arrival of each record, once this window is partitioned, each parallel instance independently changes the counter at the arrival of a record. Since every instance runs in a separate thread, there is no way to determine the order in which instances update the counter. Therefore, two partitions trying to increment a globally declared counter can create inconsistencies so that increments get overwritten.

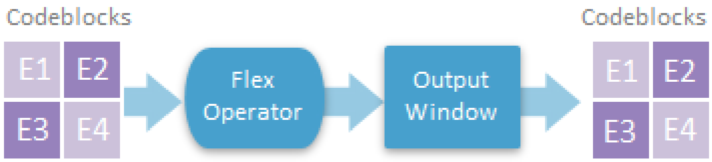
Multiple Inputs

You can partition flex operators which have multiple inputs. In such cases, if you do not specify a partitioning function, one of the inputs is automatically broadcast to the partitioned flex instances.

Transactions

Partitioning flex operators has major implications for transactions. Transactions are automatically broken down into smaller subtransactions. If one of these subtransactions fails, the others do not automatically fail with it. This can change the transaction semantics and lead to inconsistent results.

The diagram below shws a non-partitioned scenario where the transaction consists of four records. The records are processed as one block by the flex operator and an output transaction is produced that consists of these four records.



In a flex operator, you can write code to start and end a transaction. In the partitioned case, these start and end codeblocks are called within each partition for each created subtransaction. If output events are created in the start and end transaction blocks, this can lead to problems such as key duplications, which can easily be created by partitioning. Since such duplications lead to rollbacks of transactions, whole subtransactions are rolled back.

In the non-partitioned case, there are no issues because only one event (start, start_trans) is created for each transaction. In the partitioned case, each partition creates one of these events which results in duplicate keys and one of the subtransactions is rolled back.

If you wish to partition a flex operator, do not write records in the transaction start and end blocks.

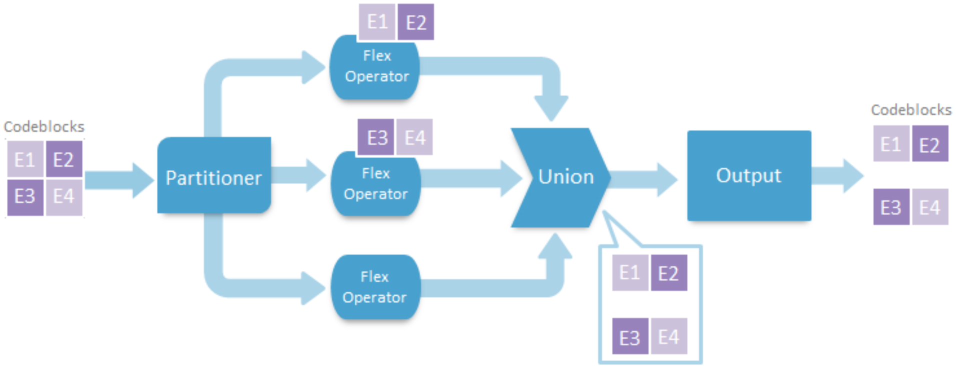
```
CREATE FLEX w3
    IN w2_out
    OUT OUTPUT WINDOW w3_out
    SCHEMA s1
    Primary Key (c1) KEEP 10 ROW
//    PARTITION BY w2_out HASH(c1) PARTITIONS 2
BEGIN
    ON w2_out
    {
        output setOpcode([c1= w2_out;|  c2 = 'event';], upsert);
    };
    on start transaction {
       output setOpcode([c1=start;|  c2 = 'start_trans';], upsert);
        };
END;
```

Pulsed Output

You can use pulsed output (EVERY X SECONDS) in flex operators to send events every X seconds instead of sending events with every new input record. The example of the partitioned case below shows how doing this can lead to problems. Each of the partitions sends events after 1 second. The partitions needs to ensure that the events produced every second do not collide with each other and create duplicate key records.

```
CREATE FLEX w3
    IN w2_out
    OUT OUTPUT WINDOW w3_out
    SCHEMA s1
    Primary Key (c1) KEEP 10 ROW
//    PARTITION BY w2_out HASH(c1) PARTITIONS 2
BEGIN
    ON w2_out
    {
        //collect records
    };
    EVERY 1 SECONDS{
        //i.e. output averages per group
    }
END;
```

Here is an example scenario that uses partitioning:



```
Create Schema TradeSchema (
   Ts INTEGER,
   Symbol STRING);

Create Schema TradeTotal (
   Symbol STRING,
   Counter INTEGER);

CREATE  INPUT  WINDOW TradeWindow
    SCHEMA TradeSchema
    PRIMARY KEY (Ts) keep 5 minutes;
```

```
CREATE FLEX FlexStateManager IN TradeWindow
     OUT OUTPUT WINDOW FlexOutput
    SCHEMA TradeTotal
    Primary Key (Symbol) KEEP 10 ROW
    PARTITION
        BY TradeWindow HASH(Symbol)
    PARTITIONS 3
BEGIN
    declare
     integer mycounter:=0;
     dictionary(string, integer) counterMap;
     typeof(TradeWindow) my;
    end;
    ON TradeWindow
    {
        if(isnull(counterMap[TradeWindow.Symbol])){
              counterMap[TradeWindow.Symbol]:=1;
          }
          else{
              print('counter..
',to_string(counterMap[TradeWindow.Symbol]));

counterMap[TradeWindow.Symbol]:=counterMap[TradeWindow.Symbol]+1;
          }
    };
    EVERY 1 SECONDS{
        for (k in counterMap) {
              output setOpcode([ Symbol=k;| Counter=counterMap[k]],
upsert);
          }
    };
END;
```

If the input events are inserts, for example, (0, SAP), (1, SAP), (1, SAP), (3, XY), (4, XY), (wait for 1 second), (5, XY), then the produced output would be (SAP, 3), (XY, 2), and after one second (SAP, 3) and (XY, 3). The dictionary in each partition stores local data that has been counted. Keep in mind that each partition is independent from each other.

Iterators

Normally, an iterator iterates over a complete input window. In the case of partitioning, the iterator is only able to iterate over the subset of events that are sent to a specific partition. Therefore, to create meaningful output in a partitioned case, ensure the use of iterators is in line with the defined partitioning key.

Guidelines for Partitioning SPLASH

General guidelines, tips, and examples of partitioning elements using SPASH logic.

Local Variables

Each partition has its own locally declared variables. Therefore, a counter of all input records only counts records that arrive at a certain partition. If you wish to use a locally declared counter, it is recommended that you use it in line with the given partition key and that you verify the results of the partitioned element.

Globally Declared Variables

Globally declared variables can also be tricky. Multiple partitions in parallel can change a globally declared variable in parallel and race-conditions cannot be avoided. For example, if a window has a user-defined method which increases a global counter variable by 1 at the arrival of each record, once this window is partitioned, each parallel instance independently changes the counter at the arrival of a record. Since every instance runs in a separate thread, there is no way to determine the order in which instances update the counter. Therefore, two partitions trying to increment a globally declared counter can create inconsistencies so that increments get overwritten.

Event Caches, Dictionaries, Vectors, and Methods

When using complex data structures like event caches, dictionaries, and vectors, be aware that partitioning may change the semantics of your original usage and lead to unexpected results.

For example, if an event cache is not defined with a key, then its original intent is to keep all records for the given time period or the limit of the number of records. However, with partitioning, records in the original event cache are distributed to several parallel event caches. Therefore, if it was required to calculate a sum() over the original event cache, partitioning would generate sum() values for each parallel event cache and as a result, break the original semantics.

Both event caches and partitioning have a concept of key. For event caches, a key defines how records in an event cache are distributed across buckets. For partitioning, a key defines how incoming records are distributed over several parallel instances. Whenever the keys for the event cache and partitioning do not match and you partition an element (stream or window) with an event cache, this can change its original semantics of the partitioned element.

Even if the event cache key matches with the partitioning key, it may still generate unexpected results with partitioning when the event cache applies a row-based retention policy and the coalesce flag is not set (see Example 1). Without setting the coalesce flat, an event cache treats records for INSERT, UPDATE, and DELETE with the same key as distinct records. When the event cache is partitioned, the same KEEP policy is used in all parallel instances. As a result, there will be n*k (where n is the number of partitions and k is the number of rows kept by the event cache) records kept by all the parallel event caches, as opposed to k records in the case of no partitioning.

Example 1:

```
CREATE input stream priceW schema (ts timestamp, isin string, price
float);
CREATE output stream maxPriceW schema (ts timestamp, isin string,
price float)
declare
        eventCache(priceW[isin], 10 events) cache;
end
```

```
PARTITION
    by priceW HASH(isin)
PARTITIONS 2
as
SELECT priceW.ts, priceW.isin isin, avg(cache.price) price FROM
priceW;
```

Additionally, user-defined methods or flex elements can perform more complicated operations on an event cache. For example, you can use the keyCache() function to access any other cache bucket independent of your current key. Partition these elements carefully since this can also alter the intended semantics. In the example below, the `Symbol` and `Symbol2` attributes of `QTrades` are in the same domain. When the flex `TradesFlex` element is hash partitioned by its input, `QTrades`, according to `Symbol,` it is possible that the bucket for `Symbol2` of a record to reside in a different parallel instance and therefore, the keyCache() statement would not be able to retrieve that bucket in the current partition.

Example 2:

```
CREATE FLEX TradesFlex
    IN QTrades
OUT OUTPUT WINDOW Top3Trades SCHEMA TradesSchema PRIMARY
KEY(Symbol,Price)
PARTITION
    BY QTrades HASH (Symbol)
PARTITIONS 3
    BEGIN
        DECLARE
            eventCache(QTrades[Symbol], manual, Price asc)
tradesCache;

        ON QTrades {
            keyCache( tradesCache, [Symbol=QTrades.Symbol2;|] );
            typeof(QTrades) rec := insertIntoCache( QTrades );
            if(rec.Id) {

                if(rec.Id <> QTrades.Id) {
                    output setOpcode(rec, delete);
                }
                output setOpcode(QTrades, upsert);
            }
        };
END;
```

Similar situation can happen for other complex data structure like dictionaries and vectors. In general, if you are writing any SPLASH logic for streams or windows that are partitioned, be aware that the logic functions only on a subset of the data and therefore, the partitioned results may be different from non partitioned streams or windows.

Guidelines for Partitioning Elements with Retention Policies

General guidelines, tips, and examples of partitioning elements which use retention policies.

A retention policy specifies the maximum number of rows or the maximum amount of time that data is retained in a window. If you partition a window for which you have previously specified a retention policy, the window retains the policy.

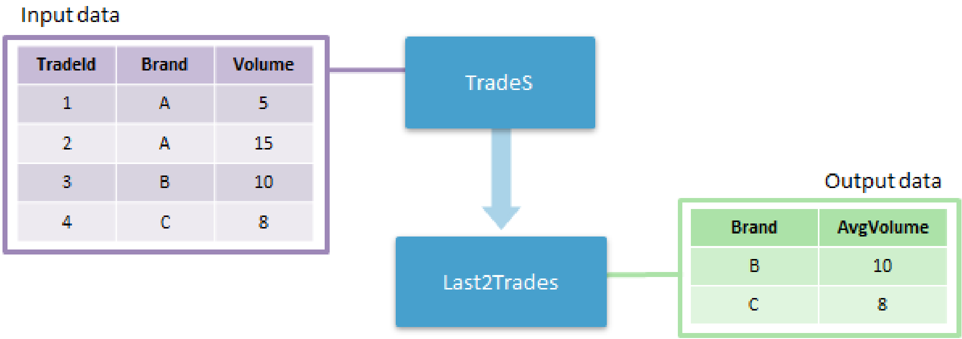
An exception to this behavior is row-based retention because the state maintained by the ESP project is larger. For example, in a non-partitioned project, specifying a retention policy for N rows stores exactly N rows. However, if using stream partitioning and you create K number of partitions, each of these partitions stores N rows.

If you specified a retention policy for a union stream, which serves as a connection point for the other downstream elements, the policy of N rows is preserved. However, there are no guarantees as to which rows will be stored in the union because the policy stores different row sets based on the order in which the events arrive at the union.

Here is a non-partitioned scenario where the window stores the exact number of specified rows (2 rows in this example):

```
CREATE INPUT WINDOW Trades SCHEMA (TradeId long, Brand string, Volume
integer)
PRIMARY KEY (TradeId);

CREATE OUTPUT WINDOW Last2Trades SCHEMA (Brand string, AvgVolume
integer)
PRIMARY KEY DEDUCED
KEEP 2 ROWS
AS
SELECT T.Brand, avg(T.Volume) as AvgVolume FROM Trades as T GROUP BY
(T.Brand);
```



Input data

TradeId	Brand	Volume
1	A	5
2	A	15
3	B	10
4	C	8

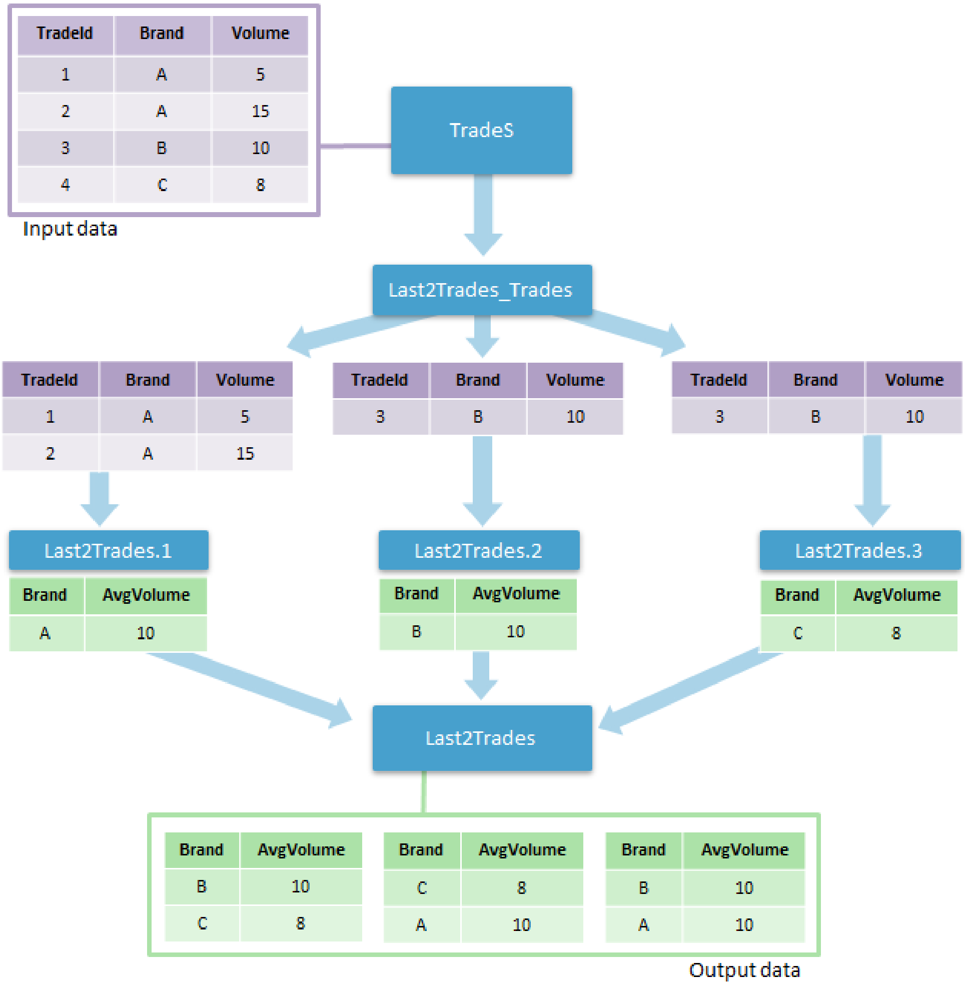
Output data

Brand	AvgVolume
B	10
C	8

Once the output window is partitioned, each partition stores the specified number of rows (<partitionCount>*2 rows in this example). The union also carries an automatically derived retention policy of 2 rows which ensures that the same number of rows are exposed as in the non-partitioned case. However, partitions are not synchronized which may lead to different rows in the final output, and therefore skew results.

```
CREATE INPUT WINDOW Trades SCHEMA (TradeId long, Brand string, Volume
integer)
PRIMARY KEY (TradeId);

CREATE OUTPUT WINDOW Last2Trades SCHEMA (Brand string, AvgVolume
integer)
PRIMARY KEY DEDUCED
KEEP 2 ROWS
PARTITION
    BY HASH (Brand)
    PARTITIONS 3
AS
SELECT T.Brand, avg(T.Volume) as AvgVolume FROM Trades as T GROUP BY
(T.Brand);
```



TradeId	Brand	Volume
1	A	5
2	A	15
3	B	10
4	C	8

Input data

TradeId	Brand	Volume
1	A	5
2	A	15

TradeId	Brand	Volume
3	B	10

TradeId	Brand	Volume
3	B	10

Brand	AvgVolume
A	10

Brand	AvgVolume
B	10

Brand	AvgVolume
C	8

Brand	AvgVolume
B	10
C	8

Brand	AvgVolume
C	8
A	10

Brand	AvgVolume
B	10
A	10

Output data

Guidelines for Partitioning Aggregation

General guidelines, tips, and examples of partitioning elements using aggregation.

When using HASH partitioning over any key aside from the GROUP BY key, the elements with the same value for the GROUP BY clause may not be located in the same partition and this may break the aggregation semantically.

The example below uses events that make up a part identifier for industrial parts. This identifier contains a part number and a group ID for the group to which a part belongs. For example, AA0001 is a part that belongs to the group AA (for example, engine) and has the part number 0001.

For example, to count all parts of a certain group, specify `GROUP BY ( left(T.PartId,2) )`. To ensure that the aggregation result is correct, you would have to send all the elements with the same group ID (as indicated by the two leftmost character of the PartId) to the same partition. However, this is not possible because you would have to specify `PARTITION BY HASH ( left(PartId,2))`.

```
CREATE INPUT WINDOW Trades SCHEMA (TradeId long, PartId string,
Volume integer)
PRIMARY KEY ( TradeId );

CREATE OUTPUT WINDOW TradeVolumePerGroup SCHEMA (PartGroup string,
Volume integer)
PRIMARY KEY DEDUCED
PARTITION
     BY HASH ( PartId )
     PARTITIONS 2
AS
SELECT left(T.PartId,2) as PartGroup, sum(T.Volume) as Volume FROM
Trades as T GROUP BY ( left(T.PartId,2) );
```

You can work around this limitation by introducing a new column through an intermediate stream which represents the `GROUP BY KEY` explicitly and can, therefore, be used as a HASH key.

```
CREATE INPUT WINDOW Trades SCHEMA (TradeId long, PartId string,
Volume integer)
PRIMARY KEY ( TradeId );

CREATE OUTPUT WINDOW Trades1 SCHEMA (TradeId long, PartId string,
PartGroup string, Volume integer)
PRIMARY KEY ( TradeId )
AS
SELECT T.TradeId as TradeId, T.PartId as PartId, left(T.PartId,2) as
PartGroup, T.Volume as Volume FROM Trades as T;

CREATE OUTPUT WINDOW TradeVolumePerGroup SCHEMA (PartGroup string,
Volume integer)
PRIMARY KEY DEDUCED
PARTITION
     BY HASH ( PartGroup )
     PARTITIONS 2
AS
SELECT T.PartGroup as PartGroup, sum(T.Volume) as Volume FROM Trades1
as T GROUP BY ( T.PartGroup );
```

Guidelines for Partitioning Modules

General guidelines, tips, and examples of partitioning modules.

You can partition an entire module when loading it (using the LOAD MODULE statement) by specifying which stream you wish to partition and the partitioning method. You can only partition a module by its input streams or windows. Mention them only once, even if they are mapped to multiple internal module streams or windows.

A module can contain a partitioned stream or load other partitioned modules. However, be aware that the total amount of partitions that your ESP project has overall can quickly grow. Specifically, if you partition a module, all internal module partitions are partitioned again when loading the module. For example, if a module partitions a stream by 5 and the module is partitioned by 4, you would have 20 parallel instances of the stream that the module partitions.

If you create too many partitions, the ESP Server may not be able to start properly. It is recommended that the number of partitions within your ESP project stays below the number of processors on your machine.

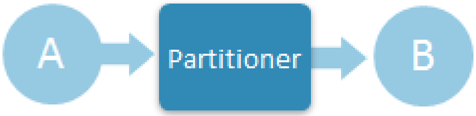
It is not possible to partition a stream that has more than one downstream connection. As a result, ESP Server inserts a copy of the partitioned stream between the stream and the partitioner. This copy does the distribution to all the successor nodes. The ID of this copied node is suffixed by .clone.

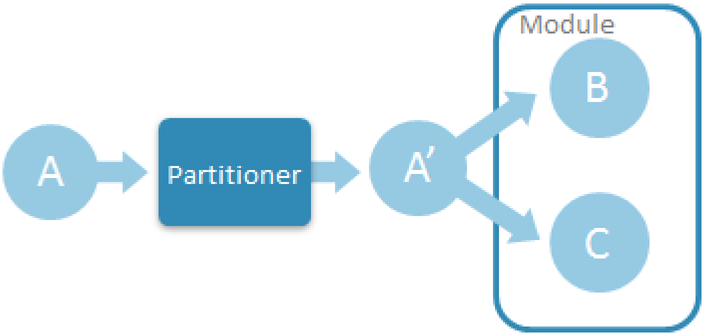
The example below displays the principle of partitioning modules. You can find the complete and running example in `$ESP_HOME/examples/ccl/SubmodulesPartitioned`. Ensure you specify the PARTITION BY clause last in the LOAD MODULE clause.

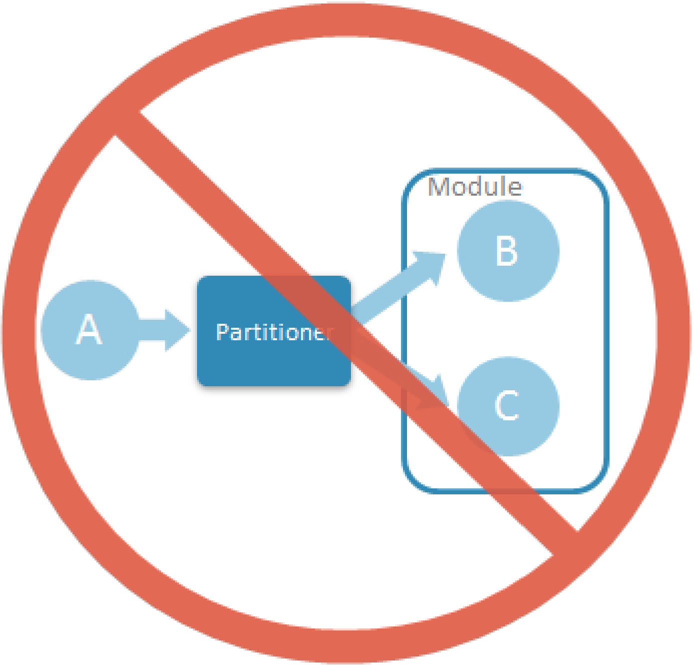
```
import 'module1.ccl';

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

LOAD MODULE Module1 AS Module1_instance_01
   IN rawStockFeed = InStocks
   OUT infoByStockSymbol = CompStocks2
   Parameters myparam = 1000
   STORES store1 = MyStore1
   PARTITION
BY InStocks HASH (Ts)
   PARTITIONS 3;
```







Guidelines for Partitioning Joins

General guidelines, tips, and examples of partitioning joins.

It is recommended that you partition joins using only HASH or CUSTOM partitioning methods as the ROUNDROBIN partitioning method typically produces inconsistent join results. You can specify a partitioning method for only a subset of the join inputs with data from the other join inputs being broadcast to all parallel join instances. For example, if a join element has three input windows but you only specify HASH partitioning for Input1, then the other two inputs (Input2 and Input3) are broadcast by default.

Choosing a Partitioning Key

When using the HASH partitioning method, it is recommended that you use a hash key for each join input that is a subset of the join key for this input. For example, for join window InputWindow1, the join conditions are InputWindow1.key1 and InputWindow1.key2. The join keys for InputWindow1 are InputWindow1.key1 and InputWindow1.key2, so it is recommended that you choose a hash key for InputWindow that is either InputWindow1.key1 or InputWindow1.key2. Using different logic can break the original join semantics and produce unexpected results.

Similarly, when using the CUSTOM partitioning method, design the partitioning logic so that it is in line with the join keys. For example, for the join window below, you can only specify a partition method on W1 or W2. For W1, meaningful hash keys are (Key1W1), (Key2W1), and (Key1W1, Key2W1). For W2, meaningful hash keys are (Key1W2), (Key2W2), and (Key1W2, Key2W2). If you use HASH partitioning for both W1 and W2, then meaningful hash key pairs are (Key1W1) - (Key1W2), (Key2W1) - (Key2W2), and (Key1W1, Key2W1) - (Key1W2, Key2W2).

```
CREATE INPUT WINDOW W1 SCHEMA (Key1W1 integer, Key2W1 string, Val1W1
integer, Val2W1 string) PRIMARY KEY (Key1W1, Key2W1);
CREATE INPUT WINDOW W2 SCHEMA (Key1W2 integer, Key2W2 string, Val1W2
integer, Val2W2 string) PRIMARY KEY (Key1W2, Key2W2);

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

Effects of Partitioning on Outer Joins

Although partitioning is generally supported for all types of joins (inner, outer, left, right, and full), partitioning the outer side of an outer join may change the original join semantics and lead to different results than a nonpartitioned scenario.

Here is an example of a left outer join:

```
CREATE INPUT WINDOW W1 SCHEMA (KeyW1 integer, ValW1 integer) PRIMARY
KEY (KeyW1);
CREATE INPUT WINDOW W2 SCHEMA (KeyW2 integer, ValW2 integer) PRIMARY
KEY (KeyW2);

CREATE OUTPUT WINDOW OW1
PRIMARY KEY (KeyW1)
PARTITION
    BY W2 HASH (KeyW2)
PARTITIONS 2
AS SELECT W1.*, W2.*
FROM W1 OUTER JOIN W2 ON W1.KeyW1 = W2.KeyW2;
```

The input data of W1 is: `<W1 ESP_OPS="i" KeyW1=”1”, ValW1=”100”>`

The input data of W2 is: `<W2 ESP_OPS="i" KeyW2=”1”, ValW2=”200”>`

Given this input data for W1 and W2, the join result in a non-partitioned case would be `<W1 ESP_OPS="i" KeyW1=”1”, ValW1=”100”, KeyW2=”1”, ValW2=”200”>`.

However, because the default partitioning method for W1 is broadcast and W1 is the outer side of the given left outer join, record `<W1>` is sent to both instances of the join in the partitioned case. The join instance, which also receives the record from W2, then produces the result record `<OW1 ESP_OPS="i" KeyW1=”1”, ValW1=”100”, KeyW2=”1”, ValW2=”200”>`. The other instance also produces the result record `<OW1 ESP_OPS="i" KeyW1=”1”, ValW1=”100”>`. Since `W1.KeyW1` is the primary key of the join window, depending on which of the two results arrive last at the final implicit union node, the record contained in the final join window can be either `<OW1 ESP_OPS="i" KeyW1=”1”, ValW1=”100”, KeyW2=”1”, ValW2=”200”>` or `<OW1 ESP_OPS="i" KeyW1=”1”, ValW1=”100”>`.

Similar scenarios can also occur for right and full joins.

Guidelines for Partitioning Elements on Log Stores

Guidelines for sizing a log store when partitioning elements that are within a log store.

The rules below assume that the log store is dedicated to a partitioned stream. For non-join cases, the rules are:

- When there is no retention specified, the sizing calculation is 2 * N where N is the size allocated for the stream in the non-partitioned case.
- When there is retention on the target, the sizing calculation is N * (K + 1) where N is the size allocated for the non-partitioned case and K is the number of partitions. This is because each partition honors the retention clause.

In the case of joins, also include the space required to store the input to the join in the log store. This means that the sizing calculation is N when you are partitioning the input, and N * K when you are broadcasting the input where N is the size required to store the input.

Modularity

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

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

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

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

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

The **CREATE MODULE** statement creates a module that can be loaded multiple times in a project, where its inputs and outputs can be bound to different parts of the larger project. The **LOAD MODULE** statement allows reuse of a defined module one or more times throughout a project. Modularity is particularly useful when used with the **IMPORT** statement, which allows you to use **(LOAD)** modules created in a separate CCL file.

Note: All module-related compilation errors are fatal.

Module Creation and Usage

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

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

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

Example: Creating and Using Modules

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

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

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

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

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

```
CREATE MODULE FilterByPrice IN TradeData OUT FilteredTradeData
BEGIN
    IMPORT 'schemas.ccl';

    CREATE INPUT STREAM TradeData SCHEMA TradesSchema;
    CREATE OUTPUT STREAM FilteredTradeData SCHEMA TradesSchema
    AS SELECT * FROM TradeData WHERE TradeData.Price > 1.00;
END;
```

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

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

```
IMPORT 'schemas.ccl';

CREATE INPUT STREAM NYSEData SCHEMA TradesSchema;

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

   - The first line of the project file imports `schemas.ccl`, which allows the use of the same schema as the module.
   - The input stream `NYSEData` represents trade information from the New York Stock Exchange.
   - The **LOAD MODULE** statement loads the module, `FilterByPrice`, which is identified by the instance name of `FilterOver1`.
   - Binding the module's input stream, `TradeData`, with the input stream `NYSEData` allows information to flow from the `NYSEData` stream into the module.
   - The output of the module is exposed to the project (`NYSEPriceOver1Data`).
   - To access the output of the module, select the information from the `NYSEPriceOver1Data` stream.

Example: Parameters in Modules

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

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

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

The module definition is:

```
CREATE MODULE FilterByPrice IN TradeData OUT FilteredTradeData
BEGIN
    IMPORT 'schemas.ccl';

    DECLARE
        parameter money(2) minimumPrice := 10.00d2;
    END;

    CREATE INPUT STREAM TradeData SCHEMA TradesSchema;
    CREATE OUTPUT STREAM FilteredTradeData SCHEMA TradesSchema AS
SELECT * FROM TradeData WHERE TradeData.Price > minimumPrice;
END;
```

Binding a Parameter to an Expression

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

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

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

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

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

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

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

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

No Parameter Binding

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

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

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

Data Recovery

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

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

When using log stores:

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

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

When you shut down the server normally, it performs a quiesce and checkpoint before it shuts down. It is therefore able to store all data currently in the project, as the data has been fully processed and is in a stable state. When an abnormal system shutdown occurs between checkpoints, there is no way of knowing the state of the system or where the uncheckpointed data was. Therefore, the uncheckpointed data on the input windows attached to log stores is replayed by streaming events down the project as though they were going through the input windows for the first time. The uncheckpointed data is replayed in an attempt to attain a state as close as possible to the state of ESP before the abnormal shutdown.

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

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

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

Log Store Optimization Techniques

Specify persistence to optimize data models for maximum performance.

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

Error Streams

Error streams gather errors and the records that caused them.

Description

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

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

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

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

Schema

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

Error Codes and Corresponding Values

- NO_ERR - 0
- GENERIC_ERROR - 1
- FP_EXCEPTION - 2
- BADARGS - 3
- DIVIDE_BY_ZERO - 4
- OVERFLOW_ERR - 5
- UNDERFLOW_ERR - 6
- SYNTAX_ERR - 7

Limitations

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

- Only errors that occur during record computation are captured in error streams. Errors in computations that occur at server start-up, such as evaluation of expressions used to initialize variables and parameters, are not propagated to error streams. Other errors, such as connection errors and noncomputational errors, are not captured in error streams.
- Errors occurring during computations that happen without a triggering record, such as in the `ON START TRANS` and `ON END TRANS` blocks of a flex block, propagate an error record where the errorRecord field contains an empty record.

- For the recordDataToRecord built-in, the stream name must be a string literal constant. This limitation is so that a record type of the return value of the built-in can be determined during compilation.
- The triggering record must be retrieved using provided built-ins. No native nested record support is provided to refer to the record directly.
- The triggering record reported is the immediate input for the stream in which the error happened. This may be a user-defined stream or an intermediate stream generated by the compiler. When using the recordDataToString and recordDataToRecord built-ins, the first argument must match the intermediate stream if one has been generated.
- The subscription utility does not automatically decrypt (convert from binary to ASCII) the error record.
- Output adapters do not automatically decrypt (convert from binary to ASCII) the error record.
- Arithmetic and conversion errors occurring in external functions (C and Java) are not handled; such errors are the users responsibility.
- Error streams are not guaranteed to work within the debugger framework.

Monitoring Streams for Errors

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

Process

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

Examples

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

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

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

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

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

```
CREATE OUTPUT ERROR STREAM vwapErrors ON DerivedStream3;
```

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

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

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

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

CHAPTER 7 Zero Data Loss

Zero data loss protects a project against data loss in the event of a client failure, server failure, or both. Achieving zero data loss requires a judicious use of log stores when you set up the project, as well as configuration of project options and clients that use guaranteed delivery (GD).

With zero data loss,

- Event Stream Processor recovers windows protected by one or more log stores to a consistent state as of the most recent checkpoint. (Any uncheckpointed data is lost and must be sent again by the publisher.)
- Clients can be confident they will not miss any events.
- Clients can minimize the number of duplicate events they receive by controlling how frequently they issue GD commits.
- Publishers can ensure that the data they publish is fully processed by the server and thereby reduce transmission of duplicates when the server restarts.
- You can optionally configure the server to control how frequently it issues automatic checkpoints and thus control how much uncheckpointed data is liable to be lost on a server failure.
- At the expense of performance, you can minimize (but not fully eliminate) the production of duplicate rows on server or subscriber restart by tweaking how frequently the server checkpoints data and how frequently GD subscribers issue GD commits.

Consistent Recovery

Consistent recovery lets you set up a project that can recover its data if it is interrupted by a server or connection failure.

The consistent recovery feature can restore all the windows in a project to a consistent state after a server or connection failure. (Recovery consistency depends on following guidelines for log stores.) When consistent recovery is enabled, the server uses coordinated checkpoints to save data in log stores. When any log store fails to complete a checkpoint, all the log stores for that project roll back to their state as of the previous successful checkpoint. This rule ensures that even if a server or connection fails, all log stores in a project are consistent with one another. However, any input data that has not been checkpointed is not recovered upon restart.

You enable consistent recovery in the project configuration (CCR file), either in Studio or by manually editing the CCR file. See the *Studio Users Guide* for details.

Enabling consistent recovery has no effect if there are no log stores in the project. When you enable consistent recovery for a project, place the log stores on a shared drive where all the machines in the Event Stream Processor cluster have access to them.

In consistent recovery mode, a project treats commits issued by publishers as checkpoint requests. When the publisher receives the return of a commit from the project, it can notify its source that the data in question has been processed.

All guaranteed delivery subscribers to a window stored in a log store receive checkpoint notifications. GD subscribers can use this notification as an indication that it is safe to commit data in its target.

Consistent recovery works well with projects configured for cold failover if log stores are set up following the log store guidelines. When a project set for cold failover stops responding, the cluster restarts the project, typically on another host. Consistent recovery enables the restarted project to come back up to a consistent state corresponding to the last checkpoint. SAP does not recommend using consistent recovery with HA active-active mode (dual project instances) because there is no guarantee that the data produced in the primary instance is identical to the data in the secondary instance. This is a consequence of the nondeterministic nature of Event Stream Processor.

When consistent recovery is not enabled (which is the default state), the project does not ensure that all the log stores recover to the same point in time after a server failure. Some log stores may recover to a checkpoint state earlier in time than other log stores because the checkpoints across log stores are not treated as an atomic operation. When there is only one log store in the project, this is not an issue.

When you use consistent recovery, the recommendation that all input windows in a project and their direct or indirect dependents be placed in the same log store no longer applies. On the contrary, SAP recommends that you use multiple log stores placed on different disks to improve performance. Using multiple log stores is possible because consistent recovery ensures that all the log stores in the project are always consistent with each other.

See also

- *Auto Checkpoint* on page 96
- *Guaranteed Delivery* on page 98
- *Achieving Zero Data Loss* on page 100
- *Log Store Guidelines* on page 104

Auto Checkpoint

Zero data loss relies on data being checkpointed (registered and saved in the project's log stores). Auto checkpoint lets you configure the checkpoint interval—the number of input transactions that triggers a checkpoint.

Checkpoints are triggered when:

- A publisher in a client application issues a commit (if the consistent recovery project option is enabled)
- The project server determines a checkpoint is required
- The project processes the number of transactions you specified in the auto checkpoint option
- The project shuts down cleanly
- The project restarts after an unexpected shutdown

Auto checkpoint lets you control how often log store checkpoints occur across all input streams and windows in the project. More frequent checkpoints mean less data is lost if the server crashes. At the maximum checkpoint frequency of every input transaction (value of 1), all input data is protected except the data from the last transaction, which might not be checkpointed before a crash. When you set checkpoint frequency, you make a trade-off: with frequent checkpoints you can reduce the amount of data at risk, but performance and latency may suffer as a result. The alternative is to increase performance but risk a larger amount of data loss by setting infrequent checkpoints.

Setting auto checkpoint guarantees that a checkpoint occurs at least every N rows where N is the checkpoint interval. The checkpoint itself may include more input rows because the system ensures that all inputs (other than the input stream that triggered the checkpoint) have consumed all the data in its input queues. The actual checkpoint may happen earlier than called for by the auto checkpoint interval if the system decides it is necessary.

When the server completes a checkpoint, it sends checkpoint messages to GD subscribers to notify them that all data up to the sequence number specified in the checkpoint message can be safely recovered by the server on restart.

Setting auto checkpoint has no effect if there are no log stores in the project. Auto checkpoint is not dependent on consistent recovery; you can use it with consistent recovery enabled or disabled.

Note: SAP recommends that you do only one of the following:

- Enable auto checkpoint.
- Configure publishers sending data to the project to issue commits, which trigger checkpoints.

See also

- *Consistent Recovery* on page 95
- *Guaranteed Delivery* on page 98
- *Achieving Zero Data Loss* on page 100

Guaranteed Delivery

Guaranteed delivery (GD) uses log stores to ensure that a GD subscriber registered with a GD window receives all the data processed by that window even if the client is not connected when the data is produced.

GD is supported only on windows assigned to log stores. For streams, delta streams, or windows assigned to memory stores, consider using persistent subscribe pattern (PSP). For information on PSP, see the *Studio Users Guide.*

When you enable guaranteed delivery on a window that has registered guaranteed delivery subscribers, the window stores a copy of every event it produces in its log store until all the registered guaranteed delivery subscribers acknowledge receiving the events.

Note: The window stores copies of events only if there are registered guaranteed delivery subscribers.

SAP recommends that when you enable guaranteed delivery and set up subscriptions, you also enable the consistent recovery option in the project configuration (CCR file).

You can set up a GD window in Studio or directly in the CCL file using a **CREATE WINDOW** or **CREATE FLEX** statement. Make sure to assign a log store to each GD-enabled window. Do not create GD-enabled windows inside modules—this is not supported because you cannot attach adapters or subscribe directly to elements in modules.

A window configured for GD also supports non-GD subscriptions. Enabling GD does not significantly affect the window’s performance when it has no registered GD subscribers.

You can subscribe to windows configured for GD using adapters, bindings, the SDKs, or the subscribe tool.

Adapters

Enable guaranteed delivery on the adapters that support it and configure the GD adapter parameters. See the *Adapters Guide* for information on adapter support for GD.

Bindings

Enable GD on any project bindings to ensure that data is delivered to remote projects. See the *Studio Users Guide* for details.

SDKs

In the SDKs, set up GD subscriptions so each client can receive data and checkpoint messages from your project. The client, in turn, must periodically respond to the project server with a commit message reporting the sequence number of the latest event the client has processed. The server does not need to save events once it knows that all GD clients have processed them, so it can free up their space for other uses.

See the *C SDK Guide*, the *Java SDK Guide*, and the *.NET SDK Guide* for more information.

Subscribe Tool

For testing or simple use cases, use the **esp_subscribe** tool to subscribe in GD mode. See the *Utilities Guide* for details.

Recommendations for Guaranteed Delivery Subscribers

Follow these recommendations to reduce your chances of receiving duplicate rows or inconsistent data after subscriber or server restart.

- Make sure the project you subscribe to is running in consistent recovery mode. See the *Studio Users Guide* or the *Configuration and Administration Guide* for details on setting project deployment options for consistent recovery.
- Subscribe to a window on which GD is enabled. You can identify GD-enabled windows:
  - Using the **supports_gd** command in the **esp_client** utility (see the *Utilities Guide* for more information)
  - Using commands in the SDKs (see the *C SDK Guide*, the *Java SDK Guide*, and the *.NET SDK Guide* for more information)
  - Using the _ESP_Streams metadata window—look for streams that have a nonzero entry in the supports_gd column (see the *Configuration and Administration Guide* for more information)
- Send data on to your client as it arrives or buffer it locally. Issue commits for only those messages for which you have received a checkpoint notification. If the client does not support commits, deliver only those messages that have been checkpointed by the server and cache the rest locally. This ensures that the client is always consistent with the server on a restart after a client or server failure.
- To minimize data loss, SAP recommends that you do one of the following:
  - Configure the publisher to issue commits frequently (see the *C SDK Guide*, the *Java SDK Guide*, and the *.NET SDK Guide* for more information).
  - Configure the Auto Checkpoint project deployment option to control how frequently the client receives checkpoint messages (see the *Studio Users Guide* or the *Configuration and Administration Guide* for details on setting project deployment options for consistent recovery.)
- When the ESP server sends a checkpoint message, send a commit to the client or send the buffered rows followed by a commit.
- Issue a GD commit with the last committed sequence number to ensure that the server does not send the data again the next time the server or the subscription restarts. Note, however, that if the server does not shut down cleanly, it resends committed events that were not checkpointed.

See also

- *Consistent Recovery* on page 95
- *Auto Checkpoint* on page 96

- *Achieving Zero Data Loss* on page 100

Achieving Zero Data Loss

A lost connection or a server crash can cause data produced by a project to be lost before it is delivered to a listening client. If you cannot afford to lose data, complete these tasks to configure zero data loss.

Task	For Instructions, see...
Create a project with a guaranteed delivery window or add a GD window to an existing project. You can do this in Studio or by adding **CREATE WINDOW** or **CREATE FLEX** statements to the project's CCL file.	*Studio Users Guide* *Programmers Reference*
Set up log stores for any windows in the project that cannot be recovered by an upstream provider. Review the guidelines, restrictions, and sizing instructions for log stores to ensure that your project can accurately and completely recreate its data after a restart.	*Chapter 8, Creating a Log Store* on page 103 and all subsections
Variables and SPLASH data structures (dictionaries, vectors, and event caches) do not persist in log stores and thus cannot be recovered after a failure. Use these structures with log stores only when: • You can provide logic to reconstruct the structures on restart, or • Processing will not be affected if the structures are missing after a restart.	–
Enable guaranteed delivery on any bindings associated with GD-enabled windows.	*Studio Users Guide*

Task	For Instructions, see...
Before you deploy the project, enable these options in the project configuration (CCR) file: • Failover • (Optional) Auto Checkpoint **Note:** SAP recommends that you enable Auto Checkpoint only if you do not configure publishers of inbound data to trigger checkpoints by issuing commits. • Consistent Recovery **Note:** SAP does not recommend enabling active-active HA mode for projects where consistent recovery is enabled. Active-active mode does not safeguard against data loss or data inconsistency when the project switches from the primary instance to the secondary.	*Studio Users Guide* *Configuration and Administration Guide*
Enable guaranteed delivery on the project's adapters. **Note:** Some adapters do not support guaranteed delivery. See the *Adapter Summary* in the *Adapters Guide* for information on GD support.	*Adapters Guide*
If auto checkpoint is not enabled, configure publishers sending data to your project to issue commits to trigger server checkpoints. When the commit call returns, the publisher knows that the server has checkpointed the data. **Note:** When a project receives data from more than one publisher, a commit issued by one publisher triggers checkpointing of data sent by all publishers. Configure publishers to retransmit any data for which they do not receive a commit confirmation.	*C SDK Guide* *Java SDK Guide* *.NET SDK Guide*
Set up guaranteed delivery subscriptions for client applications that consume data from your project.To confirm that subscribers have received and processed data, configure them to send GD commit calls in response to checkpoint notifications.	*C SDK Guide* *Java SDK Guide* *.NET SDK Guide*

See also

- *Consistent Recovery* on page 95
- *Auto Checkpoint* on page 96
- *Guaranteed Delivery* on page 98

CHAPTER 8 Creating a Log Store

If failover is enabled, configure a log store to capture the data that flows through a project.

Note: Log stores do not store SAP Sybase Event Stream Processor event logs (cluster logs, server logs, or project logs).

Create one log store per project. The preferred destination for log store files is the base directory where project files are stored.

1. In the CCL editor, create a log store using the **CREATE LOG STORE** statement:

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

2. For the **filename** property enter either a relative (preferred) or absolute file path for the location of the log store:

Relative path (preferred)	A relative path is relative to the ESP base directory. Using a relative path means that your log store automatically points to the base directory. Relative paths do not point to the directory stack; this means that the path does not start with a drive letter or slash (/).
Absolute path (not recommended)	An absolute path points to any location on your machine, regardless of the current working directory (base directory). For Windows systems, an absolute path begins with the drive letter; on UNIX and Solaris systems, the absolute path begins with a slash (/).

   The relative path location must be a shared disk accessible by all cluster nodes. The log store path is specified in the **filename** property within the log store definition. Using a relative path automatically places the log store under: `<base-directory>/<workspace-name>.<project-name>.<instance-number>`. You can view base directory definitions in the cluster configuration file (`<node-name>.xml`), under the controller section.

   SAP recommends that you use a relative path. To use an absolute path, first ensure that all cluster nodes can read and write to the absolute path you specify. This means that the location must be the same for all cluster nodes. You must also ensure that no two projects use the same path for the log store location. If using a shared disk is not possible, configure a strong affinity to ensure the project always runs on the same cluster node.

3. Enter appropriate values for the remaining properties in the **CREATE LOG STORE** statement.
4. Click **Compile** (F7).
5. Click **Run Project**.

Log Store Guidelines

Special considerations for using log stores.

General Guidelines

- Locate log stores on a shared drive accessible to all the machines in the cluster.
- Keep streams and windows that change at substantially different rates in different log stores. If a log store contains a large but nearly-static stream and a small but rapidly changing stream, each cleaning cycle must process large amounts of data from the static stream. Keeping streams separate optimizes cleaning cycles.
- Put into a log store any window fed by stateless elements (streams and delta streams).
- Put into a log store any window fed by more than one upstream source in the project data flow. This is necessary for recovery because the arrival order of rows is not preserved.
- Put into a log store any window that cannot produce the same result before and after a disruptive event such as a server crash, based on data replayed during the recovery process.
- Log stores use window names internally for identification. Start a new file for a log store when renaming a window it is attached to.
- Variables and SPLASH data structures (dictionaries, vectors, and event caches) do not persist in log stores and thus cannot be recovered after a failure. Use these structures with log stores only when:
  - You can provide logic to reconstruct the structures on restart, or
  - Processing will not be affected if the structures are missing after a restart.

Guidelines for Guaranteed Delivery

All the general guidelines above apply to log stores for windows with guaranteed delivery. In addition:

- Because copies of events are kept in the same log store the window is assigned to, the log store for a guaranteed delivery window must be significantly larger than the log store for a similar window without guaranteed delivery. Ensure that the log store for every guaranteed delivery window is large enough to accommodate the required events. If the log store runs out of room, the project server shuts down.
- Put into a log store any window on which GD is enabled and all input windows that feed GD windows. You can put windows located between the input and GD windows in a memory store if upon restart they can be reconstructed to exactly the same state they were

in before the server went down. If an intermediate window cannot be reconstructed to its previous state, put it in a log store.

- If consistent recovery is not enabled, put the GD windows and all their feeder windows into the same log store. Note, however, that placing many windows in the same log store adversely affects performance.
- If consistent recovery is enabled, you can employ as many log stores for your GD and feeder windows as necessary.

See also

- *Sizing a Log Store* on page 105
- *Log Store Sizing Reference* on page 109

Sizing a Log Store

Calculate the size of the log store your project requires. Correctly sizing your log store is important, as stores that are too small or large can lead to performance issues.

You will start this procedure by calculating your project's internal record size. An internal record represents a row in an Event Stream Processor window. Each row contains a fixed-size header plus a variable-size payload containing the column offsets, column data, and any optional fields. Use this formula for the calculation in step *1* on page 106:

$$\mathrm{HeaderSize}(56) + \mathrm{Offsets}(4 * \mathrm{M}) + \textstyle\sum_{1}^{M} \mathrm{PS}$$

In the formula,

M represents the number of columns
PS represents the primitive datatype size for each of the M columns

Primitive datatypes are the building blocks that make up more complex structures such as records, dictionaries, vectors, and event caches. This table gives the size for datatype.

Table 3. Primitive Datatype Sizes

Datatype	Size in Bytes	Notes
Boolean	1	
Decimal	18	
Integer	4	
Long	8	
String	1 + number of characters in the string	Estimate an average length

Datatype	Size in Bytes	Notes
Float	8	
Money(n)	8	
Date	8	
Time	8	
Timestamp	8	
BigDateTime	8	
Binary	4 + number of bytes in the binary value	Estimate an average length

Note: Guaranteed delivery (GD) logs hold events stored for delivery. If no GD logs are stored in the log store, you have the option of skipping step *1* on page 106, step *2* on page 107, and step *3* on page 107. Instead, compute the dataSize using the Playback feature in Studio or the **esp_playback** utility to record and play back real data to get a better idea of the amount of data you need to store. (See the *Studio Users Guide* for details on Playback or the *Utilities Guide* for details on **esp_playback**.) The log store reports "`liveSize`" in the server log when the project exits (with log level three or higher) or after every compaction (with log level six or higher). Use the "`liveSize`" value for the dataSize referenced in step *2* on page 107 and beyond.

1. For each window, calculate the size of an internal record. If the window supports GD, compute the size for the GD logs separately.

   For purposes of illustration, use this schema:

   ```
   CREATE SCHEMA TradesSchema AS (
          TradeId    LONG,
          Symbol     STRING,
          Price      MONEY(4),
          Volume     INTEGER,
          TradeDate  BIGDATETIME
   );
   ```

   a) Using the primitive sizes from the *Table 3. Primitive Datatype Sizes* on page 105 table, compute the column values—the total size in bytes for the datatypes in the schema. For the sample schema, assuming an average STRING length of 4, the calculation is:

   ```
   8 + (4 + 1) + 8 + 4 + 8 = 33 bytes
   ```

   b) Add the size of the offsets to the size of the column values. The offsets are calculated as (4 * M) where M is the number of columns. Plugging in the sample schema's five columns, we get:

   ```
   (4 * 5) + 33 = 53 bytes
   ```

   c) Add the size of the row header, which is always 56 bytes:

   ```
   56 + 53 = 113 bytes
   ```

d) Round up to the nearest number divisible by:

- 8 if ESP is running on a 64-bit architecture
- 4 if ESP is running on a 32-bit architecture

For a 64-bit installation, use this formula:

```
URS + (8 - (URS modulo 8))
```

where URS is the unrounded record size value you computed in step *1.c*. (For a 32-bit installation, substitute a 4 for each 8 in the formula.) Continuing with our example, where we assume ESP is running on a 64-bit machine,

```
113 + (8 - (1)) = 120 bytes
```

e) Label your result recordSize and make a note of it.

2. Estimate the maximum amount of data, in bytes, that you expect to collect in the log store. To do this you must determine the maximum number of records each window assigned to the log store will contain. If the window supports guaranteed delivery, treat the GD logs as a separate window, and for the record count use the maximum number of uncommitted rows you expect the GD logs to contain for this window. Add 1000 to this value because GD logs are purged only when there are at least 1000 fully committed events.

Next, for each window, determine the data size by multiplying the expected record count by the recordSize you computed in step *1.e* on page 107. Sum the data size for all the windows and GD logs to get the total size of the data that will be stored in the log store. Label this value dataSize.

Also sum the record counts for each window and GD log assigned to this log store and label that value recordCount.

3. To calculate the basic indexing overhead, multiply the recordCount from step *2* by 96 bytes. Add the result to the dataSize value.

4. Choose the value of the **reservePct** parameter. The required store size, in bytes, including the reserve, is calculated as:

storeBytes = dataSize * 100 / (100 - **reservePct**)

where dataSize is the value you computed in step *3*.

Round storeBytes up to the next megabyte.

5. Ensure the reserve cannot be overrun by the uncheckpointed data.

Estimate the maximum amount of uncheckpointed data that is produced when the input queues of all the streams, except source streams, are full. The records in the queues that are located early in the sequence must be counted together with any records they produce as they are processed through the project. Include the number of output records that are produced by the stream for each of its input records.

This example shows the stream queue depth set to the default of 1024, for a log that contains four streams ordered like this:

```
source --> derived1 --> derived2 --> derived3
```

a) Determine the number of records that are produced by each stream as it consumes the contents of its queue:
   - 1024 records may end up in `derived1`'s input queue. Assuming the queue produces one output record for one input record, it produces 1024 records.
   - 2048 records may end up in `derived2`'s input queue (1024 that are already collected on its own queue, and 1024 more from `derived1`). Assuming that `derived2` is a join and generates on average 2 output records for each input record, it produces 4096 records ([1024 + 1024] * 2).
   - 5120 records may end up in `derived3` (1024 from its own queue and 4096 from `derived2`). Assuming a pass-through ratio of 1, `derived3` produces 5120 records.

   When the project's topology is not linear, you must take all branches into account. The pass-through ratio may be different for data coming from the different parent streams. You must add up the data from all the input paths. Each stream has only one input queue, so its depth is fixed, regardless of how many parent streams it is connected to. However, the mix of records in each queue may vary. Assume the entire queue is composed from the records that produce that highest amount of output. Some input streams may contain static data that is loaded once and never changes during normal work. You do not need to count these inputs. In the example, `derived2` is a join stream, and has static data as its second input.

b) Calculate the space required by multiplying the total number of records by the average record size of that stream.
   For example, if the records in `derived1` average 100 bytes; `derived2`, 200 bytes; and `derived3`, 150 bytes, the calculation is:

   (1024 * 100) + (4096 * 200) + (5120 * 150) = 1,689,600

   Trace the record count through the entire project, starting from the source streams down to all the streams in the log store. Add the data sized from the streams located in the log store.

c) Multiply the record count by 96 bytes to calculate the indexing overhead and add the result to the volume in bytes:

   (1024 + 4096 + 5120) * 96 = 983,040

   1,689,600 + 983,040 = 2,672,640

   Verify that this result is no larger than one quarter of the reserve size:

   uncheckpointedBytes < storeBytes * (**reservePct** / 4) / 100

   If the result is larger than one quarter of the reserve size, increase the reserve percent and repeat the store size calculation. Uncheckpointed data is mainly a concern for smaller stores. Other than through the uncheckpointed data size, this overhead does not significantly affect the store size calculation, because the cleaning cycle removes it and compacts the data.

6. When you create the log store, place storeBytes, the log store size value you arrive at here, in the **CREATE LOG STORE** statement's **maxfilesize** parameter.

See also

- *Log Store Guidelines* on page 104
- *Log Store Sizing Reference* on page 109

Log Store Sizing Reference

Set sizing parameters for a log store in a **CREATE LOG STORE** statement in the project's CCL file.

The **CREATE LOG STORE** parameters described here control the size and behavior of the log store.

maxfilesize *Parameter*

The maximum file size is the largest size, in bytes, that the log store file is allowed to reach. See *Sizing a Log Store* for instructions on calculating this value.

Unlike memory stores, log stores do not extend automatically. Sizing log stores correctly is important. A store that is too small requires more frequent cleaning cycles, which severely degrades performance. In the worst case, the log store can overflow and cause processing to stop. A store that is too large also causes performance issues due to the larger memory and disk footprint; however, these issues are not as severe as those caused by log stores that are too small.

reservePct *Parameter*

The reserve is intermediate or free space maintained in every log store. It is used when the store is resized and during periodic cleaning of the store. The **reservePct** value is a percentage of the size of the log store.

Note: If the reserve space is too small and the project runs until the store fills with data, a resize attempt may cause the store to become wedged. This means that it cannot be resized, and the data can be extracted from it only by SAP Technical Support. It is safer to have too much reserve than too little. The default of 20 percent is adequate in most situations. Multigigabyte stores may use a reduced value as low as 10 percent. Small stores, under 30MB, especially those with multiple streams, may require a higher reserve (up to 40 percent). If you find that 40 percent is still not enough, increase the size of the store.

Event Stream Processor automatically estimates the required reserve size and increases the reserve if it is too small. This usually affects only small stores. It is a separate operation from resizing the log store itself, which must be performed by a user.

Note: Increasing the reserve reduces the amount of space left for data. Monitor server log messages for automatic adjustments when you start a new project. You may need to increase the store size if these messages appear.

As the store runs, more records are written into it until the free space falls below the reserve. At this point, the source streams are temporarily stopped, the streams quiesced, and the checkpoint and cleaning cycle are performed. Streams do not quiesce immediately: they must first process any data collected in their input queues. Any data produced during quiescence is added to the store, meaning that the reserve must be large enough to accommodate this data and still have enough space left to perform the cleaning cycle. If this data overruns the reserve, the store becomes wedged, because it cannot perform the cleaning cycle. The automatic reserve calculation does not account for uncheckpointed data.

Log Store Size Warnings

As the amount of data in the store grows, if the free space falls below 10 percent (excluding the reserve), Event Stream Processor starts reporting "`log store is nearing capacity`" in the server log. If the data is deleted from the store in bursts, (for example, if data is collected during the day, and data older than a week is discarded at the end of the day), these messages may appear intermittently even after the old data has been flushed. As the cleaning cycle rolls over the data that has been deleted, the messages disappear.

Unless your log store is very small, these warnings appear before the store runs out of space. If you see them, stop Event Stream Processor when convenient, and increase the store size. Otherwise, Event Stream Processor aborts when the free space in the project falls below the reserve size.

Recovering from a Wedged Log Store

If a log store is sized incorrectly, the entire reserve may be used up, which causes the store to become wedged. If this happens, you cannot resize the log store or preserve the content. Delete the store files and restart Event Stream Processor with a clean store. If you make a backup of the store files before deleting them SAP Technical Support may be able to extract content. Change the store size in the project, and it is resized on restart. You cannot decrease the store size. When you restart a project after resizing the store, it will likely produce server log messages about the free space being below the reserve until the cleaning cycle assimilates the newly added free space.

ckcount *Parameter*

The **ckcount** (checkpointing count) parameter affects the size of uncheckpointed data. This count shows the number of records that may be updated before writing the intermediate index data. Setting it to a large value amortizes the overhead over many records to make it almost constant, averaging 96 bytes per record. Setting it to a small value increases the overhead. With the count set to zero, index data is written after each transaction, and for the single-transaction records the overhead becomes:

$96 + 32 * \text{ceiling} (\log_2(\text{number\_of\_records\_in\_the\_stream}))$

If a stream is small (for example, fewer than 1000 records), the overhead for each record is:

96 + 32 * ceiling ($\log_2$(1000)) = 96 + 32 * 10 = 416

In many cases, the record itself is smaller than its overhead of 416 bytes. Since the effect is logarithmic, large streams are not badly affected. A stream with a million records has a logarithm of 20 and incurs an overhead of 736 bytes per record. The increased overhead affects performance by writing extra data and increasing the frequency of store cleaning.

If your project includes any windows configured for guaranteed delivery (GD), consider adjusting the value of **ckcount** to improve performance and latency.

sweepamount Parameter

The **sweepamount** parameter determines how much of the log file is “swept through” during each cleaning pass. It must be between 5 percent to 20 percent of the **maxfilesize** parameter. A good lower bound for the sweep size is half the size of the write cache on your storage array. Usually, it indicates a sweep size of 512 to 1024 megabytes. Smaller sweep sizes minimize spikes in latency at the expense of a higher average latency. High values give low average latency, with higher spikes when reclaiming space.

If the value of the **sweepamount** parameter is too small, the system performs excessive cleaning; in some cases, this does not allow the log store to free enough space during cleaning.

The size of the sweep is also limited by the amount of free space left in reserve at the start of the cleaning cycle. If the reserve is set lower than the sweep amount and the sweep does not encounter much dead data, the sweep stops if the relocated live data fills up the reserve. The swept newly cleaned area becomes the new reserve for the next cycle. Unless other factors override, SAP recommends that you keep the sweep and the reserve sizes close to each other. **reservePct** is specified in percent while **sweepamount** is specified in megabytes.

If your project includes any windows configured for guaranteed delivery (GD), consider adjusting the value of **sweepamount** to improve performance and latency.

Log Store Size and File Locations

Ensure the total size of all log store files does not exceed the size of the machine’s available RAM. If this occurs, the machine takes longer to process the data, causing all monitoring tools to display low CPU utilization for each stream, and standard UNIX commands such as **vmstat** to display high disk usage due to system paging.

For storing data locally using log stores, SAP recommends that you use a high-speed storage device, for example, a redundant array of independent disks (RAID) or a storage area network (SAN), preferably with a large dynamic RAM cache. For a moderately low throughput, place backing files for log stores on single disk drives, whether SAS, SCSI, IDE, or SATA.

See also

- *Log Store Guidelines* on page 104
- *Sizing a Log Store* on page 105

CHAPTER 9 Writing SPLASH Routines

Reviewing samples of SPLASH code is the best way to familiarize yourself with its constructs.

These code samples show how to use SPLASH. To see projects utilizing SPLASH that you can run on your Event Stream Processor, refer to *Using SPLASH In Projects*.

Internal Pulsing

A stock market feed is a good example of several updates flowing into a stream.

Suppose the stock market feed keeps the last tick for each symbol. Some of the downstream calculations might be computationally expensive, and you might not need to recalculate on every change. You might want to recalculate only every second or every ten seconds. How can you collect and pulse the updates so that the expensive recalculations are done periodically instead of continuously?

The dictionary data structure and the timer facility allow you to code internal pulsing. Let's suppose that the stream to control is called InStream. First, define two local variables in the Flex operator:

```
integer version := 0;
dictionary(typeof(InStream), integer) versionMap;
```

These two variables keep a current version and a version number for each record. The SPLASH code handling events from the input stream is:

```
{
  versionMap[InStream] := version;
}
```

The special Timer block within the Flex operator sends the inserts and updates:

```
{
  for (k in versionMap) {
    if (version = versionMap[k])
      output setOpcode(k, upsert);
  }
  version++;
}
```

You can configure the interval between runs of the Timer block in numbers of seconds. Only those events with the current version get sent downstream, and the version number is incremented for the next set of updates.

This code works when InStream has only inserts and updates. It's a good exercise to extend this code to work with deletes.

Order Book

One example inspired by stock trading maintains the top of an order book.

Suppose there is a stream called Bid of bids of stocks (the example is kept simple by not considering the offer side), with records of the type:

```
[integer Id; | string Symbol; float Price; integer Shares; ]
```

where Id is the key field, the field that uniquely identifies a bid. Bids can be changed, so not only might the stream insert a new bid, but also update or delete a previous bid.

The goal is to output the top three highest bids any time a bid is inserted or changed for a particular stock. The type of the output where Position ranges from 1 to 3 is:

```
 [integer Position; | string Symbol; float Price; integer Shares; ]
```

For example, suppose the Bids have been:

```
[Id=1; | Symbol='IBM'; Price=43.11; Shares=1000; ]
[Id=2; | Symbol='IBM'; Price=43.17; Shares=900]
[Id=3; | Symbol='IBM'; Price=42.66; Shares=800]
[Id=4; | Symbol='IBM'; Price=45.81; Shares=50]
```

With the next event:

```
[Id=5; | Symbol='IBM'; Price=46.41; Shares=75]
```

The stream should output the records

```
[Position=1; Symbol='IBM'; | Price=46.41; Shares=75]
[Position=2; Symbol='IBM'; | Price=45.81; Shares=50]
[Position=3; Symbol='IBM'; | Price=43.17; Shares=900]
```

Note: The latest value appears at the top.

One way to solve this problem is with an event cache that groups by stock and orders the events by price:

```
eventCache(Bids[Symbol], coalesce, Price desc) previous;
```

The following code outputs the current block of the order book, down to the level specified by the `depth` variable.

```
{
  integer i := 0;
  string symbol := Bids.Symbol;
  while ((i < count(previous.Id)) and (i < depth) ) {
    output setOpcode([ Position=i; Symbol = symbol; |
                       Price=nth(i,previous.Price);
                       Shares=nth(i,previous.Shares);
                       ], upsert);
    i++;
  }
  while (i < depth) {
    output setOpcode([ Position=i; Symbol=symbol ], safedelete);
    i++;
  }
}
```

CHAPTER 10 Integrating SPLASH into CCL

CCL uses Flex operators to execute SPLASH code to process events. They have local declaration blocks, which are blocks of SPLASH function and variable declarations. They also have one method block per input stream and an optional timer block also written in SPLASH.

Access to the Event

When an event arrives at a Flex operator from an input stream, the method for that input stream is run.

The SPLASH code for that method has two implicitly declared variables for each input stream: one for the event and one for the old version of the event. More precisely, if the input stream is named InputStream, the variables are:

- `InputStream`, with the type of record events from the input stream, and
- `InputStream_old`, with the type of record events from the input stream.

When the method for input stream is run, the variable `InputStream` is bound to the event that arrived from that stream. If the event is an update, the variable `InputStream_old` is bound to the previous contents of the record, otherwise it is null.

Note: Delete events always come populated with the data previously held in the input stream.

A Flex operator can have more than one input stream. For instance, if there is another input stream called AnotherInput, the variables `AnotherInput` and `AnotherInput_old` are implicitly declared in the method block for InputStream. They are set to null when the method block begins, but can be assigned within the block.

Access to Input Windows

Within method and timer code in Flex operators, you can examine records in any of the input windows.

More precisely, there are implicitly declared variables:

- *`<InputWindowName>`*`_stream` and
- *`<InputWindowName>`*`_iterator`.

The variable *`<WindowName>`*`_stream` is quite useful for looking up values. The *`<WindowName>`*`_iterator` is less commonly used and is for advanced users.

For example, suppose you are processing events from an input stream called Trades, with the following records:

```
[ Symbol='T'; | Shares=10; Price=22.88; ]
```

You might have another input window called Earnings that contains recent earnings data, storing records:

```
[ Symbol='T'; Quarter="2008Q1"; | Value=10000000.00;  ]
```

In processing events from Earnings, you can look up the most recent Trades data using:

```
Trades := Trades_stream[Earnings];
```

The record in the Trades window that has the same key field Symbol. If there is no matching record in the Trades window, the result is null.

When processing events from the Trades window, you can look up earnings data using:

```
Earnings := Earnings_stream{ [ Symbol = Trades.Symbol; | ] };
```

The syntax here uses curly braces rather than square brackets because the meaning is different. The Trades event does not have enough fields to look up a value by key in the Earnings window. In particular, it's missing the field called Quarter. The curly braces indicate "find any record in the Earnings window whose Symbol field is the same as `Trades.Symbol`". If there is no matching record, the result is null.

If you have to look up more than one record, you can use a `for` loop. For instance, you might want to loop through the Earnings window to find negative earnings:

```
for (earningsRec in Earnings_stream) {
  if ( (Trades.Symbol = Earnings.Symbol) and (Earnings.Value < 0) ) {
    negativeEarnings := 1;
    break;
  }
}
```

As with other `for` loops in SPLASH, the variable `earningsRec` is a new variable whose scope is the body of the loop. You can write this slightly more compactly:

```
for (earningsRec in Earnings_stream where Symbol=Trades.Symbol) {
  if (Earnings.Value < 0) {
    negativeEarnings := 1;
    break;
  }
}
```

This loops only over the records in the Earnings window that have a Symbol field equal to `Trades.Symbol`. If you happen to list the key fields in the `where` section, the loop runs very efficiently. Otherwise, the `where` form is only nominally faster than the first form.

Using a Flex operator, you can access records in the window itself. For instance, if the Flex operator is called Flex1, you can write a loop just as you can with any of the input windows:

```
for (rec in Flex1) {
  ...
}
```

Output Statement

Typically, a Flex operator method creates one or more events in response to an event. In order to use these events to affect the store of records, and to send downstream to other streams, use the `output` statement.

Here's code that breaks up an order into ten new orders for sending downstream:

```
integer i:= 0;
while (i < 10) {
  output setOpcode([Id = i; |
                    Shares = InStream.Shares/10;
                    Price = InStream.Price; ], upsert);
}
```

Each of these is an upsert, which is a particularly safe operation; it gets turned into an insert if no record with the key exists, and an update otherwise.

Notes on Transactions

A Flex operator method processes one event at a time. The Event Stream Processor can, however, be fed data in transaction blocks (groups of insert, update, and delete events).

In such cases, the method is run on each event in the transaction block. The Event Stream Processor maintains an invariant: a stream takes in a transaction block, and produces a transaction block. It's always one block in, one block out. The Flex operator pulls apart the transaction block, and runs the method on each event within the block. All of the events that `output` are collected together. The Flex operator then atomically applies this block to its records, and sends the block to downstream streams.

If you happen to create a bad event in processing an event, the whole block is rejected. For example, if you try to output a record with any null key columns.

```
output [ | Shares = InStream.Shares; Price = InStream.Price; ];
```

This whole transaction block would be rejected. Likewise, if you try the following implicit insert:

```
output [Id = 4; |
        Shares = InStream.Shares;
        Price = InStream.Price; ];
```

If there is already a record in the Flex operator with Id set to 4, the block is rejected. You can get a report of bad transaction blocks by starting the Event Stream Processor with the `-B` option.

Often it's better to ensure that key columns are not null, and use `setOpcode` to create upsert or safedelete events so that the transaction block is accepted.

Transaction blocks are made as small as possible before they are sent to other streams. For instance, if your code outputs two updates with the same keys, only the second update is sent downstream. If your code outputs an insert followed by a delete, both events are removed from the transaction block. Thus, you might output many events, but the transaction block might contain only some of them.

CHAPTER 11 Using SPLASH in Projects

Two projects demonstrate how SPLASH is used.

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 12 ESP Datatypes in SQL Queries

There are several methods of querying ESP using SQL. All SQL statements must follow these rules for dealing with the specified datatypes in ESP.

Inserting Date and Time Data

To insert date, timestamp, and bigdatetime values into a SQL statement, use the **undate**,**untime**, **untimestamp**, and **unbigdatetime** functions to convert the input values into the correct datatype.

```
insert into win
(c_key,date,time,timestamp,bigdatetime)
values(21478,undate('2013-02-11
17:35:45'),untime(17:35:45'),untimestamp('2013-02-11 17:35:45'),
unbigdatetime('2013-02-11 17:35:45'))
```

Specifying Money Data

When specifying the value for money data, you must append either `d` or `D`.

```
insert into win
(c_key,c_money)
values(111770,8.12345d)
```

There are fifteen money datatypes with a fixed number of digits following the decimal point: `money1` with one digit, `money2` with two digits, ... `money15` with fifteen digits. When specifying a value for one of these datatypes, you must append either `d` or `D` and the number following `money` (for example `d7` for `money7`.

```
insert into win
(c_key,c_money1,c_money2,c_money3,c_money4,c_money5,c_money10,c_mon
ey15)
values(111770,8.12345d1,8.12345d2,8.12345d3,8.12345d4,8.12345d5,8.1
23456789012345d10,8.123456789012345d15)
```

Reading and Writing Binary Data

To write binary data to ESP in a SQL statement, use the **hex_binary()** function to convert the string representation of the hexadecimal value to binary.

```
insert into win
(c_key,c_binary)
values(111770,hex_binary('9140ACA0361856334DD319F05'))
```

To read binary data from ESP in an SQL statement, use the **hex_string()** function to convert the binary value to a string representation of the hexadecimal value.

```
select c_key,hex_string(c_binary)from win
```

CHAPTER 13 PowerDesigner for Event Stream Processor

Event Stream Processor users create and manipulate the ESP schema using PowerDesigner®.

PowerDesigner is a powerful modeling tool. Event Stream Processor users can use it develop physical data models as well as the logical data models that define the ESP schema.

Getting Started

PowerDesigner® is a tool for creating and manipulating ESP schema. Optionally, it can be used with physical data models.

This guide is intended for database and application development staff, and for SAP Professional Services representatives, customer IT support, and other technical personnel who set up and administer PowerDesigner. It includes information you need to understand, model, and modify logical schema definitions and physical database structure when developing schema.

Data Modeling Scenarios

Integrated modeling supports efficient schema definition and database design, and consistent production deployments.

Using the ESP Schema and extensions, you can:

- Model schema in the ESP Schema model, a PowerDesigner logical data model
- Convert an ESP Schema logical data model to SAP HANA, ASE, or SAP Sybase IQ physical data models
- Convert existing SAP HANA, ASE, and SAP Sybase IQ physical data models to an ESP Schema logical data model
- Import schema definitions defined in a CCL file into an ESP Schema model
- Export schema definitions from an ESP Schema model into a CCL file
- Validate a model using custom checks for ESP Schema, in addition to the standard PowerDesigner checks
- Analyze the impact of changes to schema, a model, or a database table on all components in the integrated model

The corresponding adapter (IQ, HANA, ASE) schema must match the SAP Sybase IQ, HANA, and ASE database schema for all tables in which data is inserted. After you make changes, you can use PowerDesigner to produce a set of data definition language (DDL) statements directly from the physical data models (IQ, HANA, and ASE). PowerDesigner saves the DDL in a SQL script that you can run to generate the tables and other objects for the target databases.

DDL generation does not require use of the extended modeling feature.

Sample PowerDesigner Project

A sample project supports integrated modeling.

You can install a PowerDesigner sample project that includes:

- A sample ESP Schema logical model
- SAP Sybase IQ, SAP HANA, and ASE physical data models

Opening the Sample Project

Open the sample model from the sample project.

1. Choose **Start** > **Programs** > **Sybase** > **PowerDesigner 16**.
2. In the Welcome dialog, under Getting started, choose **Open Model or Project**.

   If you are not a first-time user, you may see different options in the Welcome dialog, based on your previous work in PowerDesigner.
3. Browse to the sample project, by default in `%PowerDesigner 16\Examples\ESP\ESP.prj`, and choose **Open**.

   PowerDesigner opens a workspace for the ESP Schema sample project.
4. Double-click the **ESP** project (💼).
   The sample project opens with the sample ESP Schema model, SAP Sybase IQ model, ASE model, SAP HANA model, and the Model Relationship Diagram in the Browser view.

Learning More About PowerDesigner

This guide tells you how to use PowerDesigner in ESP.

For more information on using PowerDesigner, press **F1** to open the online help, or see the PowerDesigner online product documentation, especially:

- ***Core Features Guide* –** PowerDesigner interface and model basics
- ***Data Modeling* –** Building, checking, and generating models and databases

To view PowerDesigner online tutorials, choose **Help** > **Tutorial Videos**.

Data Model

PowerDesigner includes a logical data model for ESP schema and three physical data models for the SAP Sybase IQ, SAP HANA, and ASE databases.

The indexes for both physical data models are database-specific and must be defined individually. You can open, view, modify, and extend the data models using PowerDesigner.

ESP Schema Logical Data Model

The ESP Schema model represents market data in a logical data model independent of any data store.

The ESP Schema logical model represents the building of schema and the databases parsing schema and storing them.

The ESP Schema model contains a definition for each schema. The schema definitions are contained in the Market Data diagram in the sample ESP Schema model. Adding schema to the diagram is optional.

To create a new ESP Schema model, you can:

- Create it from scratch using the ESP Schema Model category
- Create it from scratch using the `ESPSchema.xem` file to extend the model during or after creation
- Generate it from an SAP Sybase IQ, SAP HANA, or ASE physical data model

Finding an Object in a Diagram

Locate any object with a symbol in a diagram or among several diagrams. Objects without graphical symbols, such as domains, are not shown in diagrams.
Right-click an object in the Browser and select **Find in Diagram**.

Data Model Tables

A list of all data model tables in the Market Data diagrams with their code names and descriptions.

Table 4. Data model tables

Table name	Code	Description
Bond History	BOND_HISTORY	Stores bond historical data, one record per each trading date. The data includes daily price and yield values (open/close, high/low), trade volume (number of bonds traded), and so on, for each bond.

Table name	Code	Description
Bond Quote	BOND_QUOTE	Stores real-time (intraday) quote data. Each quote record includes a yield, bid/ask price, and size (in other words, a number of bonds offered at a bid/ask price).
Bond Trade	BOND_TRADE	Stores real-time (intraday) trade data. Each trade record includes a bond's price and yield and a transaction's size (number of bonds traded).
Dividend Event	DIVIDEND_EVENT	Stores information on a dividend payment event when a shareholder receives a certain payment for each share of stock owned. The dividend amount is commonly defined as a certain percentage of a share price but can also be specified as a monetary amount. The Monetary or Percentage Indicator (MOP_INDICATOR) column indicates how the dividend amount is defined.
Index History	INDEX_HISTORY	Stores the index’s historical data, one record per each trading date. The data includes the index’s daily values (open/close, high/low) and trade volume.
Index Intraday	INDEX_INTRADAY	Stores the index’s real-time (intraday) data that shows its value movements during a trading day. Each data point includes an index value and trade volume.
Mutual Fund History	MUTL_FUND_HIST	Stores the historical data for a mutual fund, one record per each trading date. The data includes a trade date and price.
Option History	OPTION_HISTORY	Stores the options historical data, one record per each trading date. The data includes options daily price (open/close, high/low), trade volume (number of contracts traded), and so on.
Option Quote	OPTION_QUOTE	Stores the options real-time (intraday) quote data. Each quote record includes a bid/ask price, size (number of contracts offered at a bid/ask price), and so on.
Option Trade	OPTION_TRADE	Stores the options real-time (intraday) trade data. Each trade record includes a trade's price, size (number of contracts traded), and so on.

Table name	Code	Description
Split Event	SPLIT_EVENT	Stores information on a stock split event when the number of outstanding shares of a company's stock is increased and the price per share is simultaneously decreased so that proportionate equity of each shareholder remains the same. The split is characterized by a split factor; a factor of 0.5 indicates that the number of shares is increased two times and that the share price is decreased two times. In a less common reverse split, the number of shares is decreased and the price per share is increased in a similar manner; a split factor of 2 indicates that the number of shares is decreased two times and that the share price is increased two times.
Stock History	STOCK_HISTORY	Stores the stock historical data, one record per each trading date. The data includes stocks daily prices (open/close, high/low) and trade volume (number of shares traded).
Stock Quote	STOCK_QUOTE	Stores the stocks' real-time (intraday) quote data. Each quote record includes a bid/ask price and corresponding size values (in other words, a number of shares offered at bid/ask price).
Stock Trade	STOCK_TRADE	Stores the stocks' real-time (intraday) trade data. Each trade record includes a transaction's price and size (in other words, a number of shares traded).

Extensions

Extensions (`.xem` files) provide means for customizing and extending PowerDesigner metaclasses, parameters, and generation. Extended models can be used to store additional information, or to change model behavior.

PowerDesigner provides four `.xem` files:

- **ESPSchema.xem –** extensions for an logical data model. Contains rules and code that let you model ESP Schema in a PowerDesigner logical data model.
- **IQ.xem –** extensions for a physical data model. Contains only transformation rules needed to convert an ESP Schema definition to an SAP Sybase IQ table definition, in an SAP Sybase IQ model.

- **ASE.xem –** extensions for a physical data model. Contains only transformation rules needed to convert an ESP Schema definition to an ASE table definition, in an ASE model.
- **HANA.xem –** extensions for a physical data model. Contains only transformation rules needed to convert an ESP Schema definition to an SAP HANA table definition, in a HANA model.

When you use the models provided with PowerDesigner, the extensions are present. When you create a new model using the ESP model category set, extensions are applied automatically.

When you create a new model without using the ESP Model categories, or when you have an existing model you can extend it using the PowerDesigner tools and ESP extension files.

Category Set

You can set the ESP category set to create any ESP model type.

The ESP model category set includes ESP Schema, SAP Sybase IQ, SAP HANA, and ASE categories. To create new models from this category set, you must enable the categories in PowerDesigner. You can either merge the ESP categories with others that you use, or change PowerDesigner to use only the ESP categories.

Once you set up the ESP category set, you can create any ESP model type and extend it with the appropriate extension.

The `ESP.mcc` file, installed with the extensions, defines the ESP categories.

Schema Definitions

A schema definition in the ESP Schema model represents a data stream in ESP.

The sample ESP Schema model contains a schema definition for each market data table. You can customize any schema definition, or create a new one.

To create new schema in the ESP Schema model, you can either:

- Create schema in PowerDesigner, and then generate a CCL file from it, or,
- Import schema definitions that are defined in a CCL file.

Each schema definition contains:

- **identifiers –** associate schema with columns that are keys in the associated table.
- **attributes –** associate schema with a destination column name in the SAP Sybase IQ, SAP HANA, and ASE databases with length and precision where appropriate, lookup table and column information for columns that are foreign keys, and descriptive notes.

Sample Schema Definition List

Sample schema definitions correspond to the Market Data diagram provided with PowerDesigner. While each schema appears in the SAP Sybase IQ, SAP HANA, and ASE Market Data diagram, not every table in that diagram is a schema.

- Bond History
- Bond Quote
- Bond Trade
- Dividend Event
- Index History
- Index Intraday
- Mutual Fund History
- Option History
- Option Quote
- Option Trade
- Split Event
- Stock History
- Stock Quote
- Stock Trade

Impact and Lineage Analysis

PowerDesigner provides powerful tools for analyzing the dependencies between model objects.

When you perform an action on a model object, in a single operation you can produce both:

- **Impact Analysis –** to analyze the effect of the action on the objects that depend on the initial object.
- **Lineage Analysis –** to identify the objects that influence the initial object.

These tools can help you answer questions like these:

- If I change the precision on a column in my ASE model which I generated from the ESP schema model, what table columns in my SAP Sybase IQ or SAP HANA model must also change, and what schema are affected?
- Which schema fields influence each column in my ASE, SAP HANA, and SAP Sybase IQ models?
- If I delete a column from my IQ model, what is the impact on tables and columns in my ASE and SAP Sybase IQ models, and what schema definitions must change in my ESP Schema model?

Extended Model Setup

Your installer will set up the use of extensions automatically for you.

To apply the extensions automatically for new models, set up and use the ESP Schema model category set.

To integrate existing PDMs with the ESP model, extend the models by attaching the appropriate extensions file.

Extending an Existing Model

Attach extensions to any SAP Sybase IQ, SAP HANA, or ASE physical data model, or to an logical data model that was generated from an ESP physical data model but not extended.

1. Open the model you want to extend.
2. From the PowerDesigner main menu, choose **Model > Extended Model Definitions**.

   Tip: If **Extended Model Definitions** is not in the menu, make sure that the extensions file is unzipped in the folder where PowerDesigner is installed.
3. Click **Import an Extended Model Definition** .

   A list shows available extensions that have not been applied to this model.
4. Select the correct model extension and choose **OK**.
   For example, to extend an ASE physical data model, choose **ASE**.
5. In the **List of Extended Model Definitions** dialog, choose **OK** to extend the model.

PowerDesigner applies the ESP extensions to the model. No other changes are made. For example, a generic logical data model is not transformed to an ESP Schema model simply by adding the extensions.

Setting Up the Model Category Set File

Set up PowerDesigner to use the ESP category set for new models.

PowerDesigner can display only one set of categories in the New Model dialog. While not required, using the ESP category makes it easier to develop models for use with Event Stream Processor.

Decide which option you want to use to create new models:

Option	Action required
Only the installed ESP category set	Change categories
ESP category set merged with existing categories	Merge ESP categories

Option	Action required
Neither	Manually extend any models

Merging ESP Categories

When you create a new model using categories, you can see the existing categories, as well as the three standard ESP categories. You can merge existing model categories with the ESP category.

1. Choose **Tools > Resources > Model Category Sets**.
2. From the list in the dialog, select the set you want to add to the ESP category.
3. Click the **Merge** button in the toolbar.
4. Select **ESP** from the list and choose **OK**.

Changing the Default Category

Change the default category to the ESP category, so that you can create new ESP models.

1. From the PowerDesigner main menu, choose **Tools > General Options**.
2. Under Category, select **Model Creation**.
3. In the Model Creation frame, with **Enable categories** checked, select a default category set.
4. Choose **OK**.

Setting Datatypes for an ESP Schema

Manually set the datatype attribute for a ESP Schema definition if the ESP Data Type column in the Attributes tab of an ESP schema definition is empty or shows the wrong values.

You may need to set datatypes for a logical data model you generate from a physical data model, if the generation process cannot determine how to convert the database datatype to an Event Stream Processor datatype. Datatypes for the shipped sample model are set correctly and no further adjustments are necessary.

1. Right-click a schema definition and choose **Properties**.
2. Click the **Attributes** tab and review values in the ESP Data Type column.

   For example, in the sample model, the Bond Quote Attributes shows these datatypes:

Attribute Name	ESP Datatype
Instrument	**string**
Quote Date	**date**
Quote Sequence Number	**integer**

Attribute Name	ESP Datatype
Quote Time	**timestamp**
Ask Price	**money(4)**
Ask Size	**integer**
Bid Price	**money(4)**
Bid Size	**integer**
Yield	**money(2)**

If values are missing or incorrect, continue with steps 3 - 5.

3. Click **Customize Columns and Filter** (**Ctrl+U**).
4. If needed, adjust columns available for viewing:
   a) Unselect **Data Type**, **Length**, and **Precision**.
   b) Select:

      Name
      ESP Datatype
      Length
      Precision
      Mandatory
      Primary Identifier
      Displayed (selected by default)
5. Use the controls below the list to adjust the order so that **Primary Identifier** and **Displayed** are the last two checkboxes.

Performing this task once corrects the datatypes for all schema definitions.

ESP Schema Model Development

Develop schema using the PowerDesigner extensions.

You can:

- Explore the sample model
- Create a schema model using categories, or by creating and extending a logical data model
- Add schema to models
- Validate your schema with built-in checks, as well as custom ones
- Import defined schema definitions into an ESP Schema model from CCL files
- Export schema definitions from the ESP Schema model into CCL files

Exploring the Sample Model

Review the sample model from the sample project.

Prerequisites

Install the sample model and complete the extended model setup.

Task

1. Start PowerDesigner and open the sample project with the sample model.
2. To open any of the models, either:
   - Double-click the model in the Model Relationship Diagram, or,
   - In the Browser tree, double-click the model, or right-click and choose **Open** or **Open as read-only**.

   Note: Do not save changes to the installed sample model. Save it to another folder so that a new version of the model and project are created.
3. To display the sample schema definitions in the ESP Schema model, expand the navigation buttons in the Browser tree.
4. To see more information on a schema definition:
   - Right-click the schema definition, an identifier, or an attribute in the tree view and choose **Properties**, or,
   - Right-click the schema definition in the tree view and choose **Find in Diagram**.

   Explore the SAP Sybase IQ, SAP HANA, and ASE models in the same way.

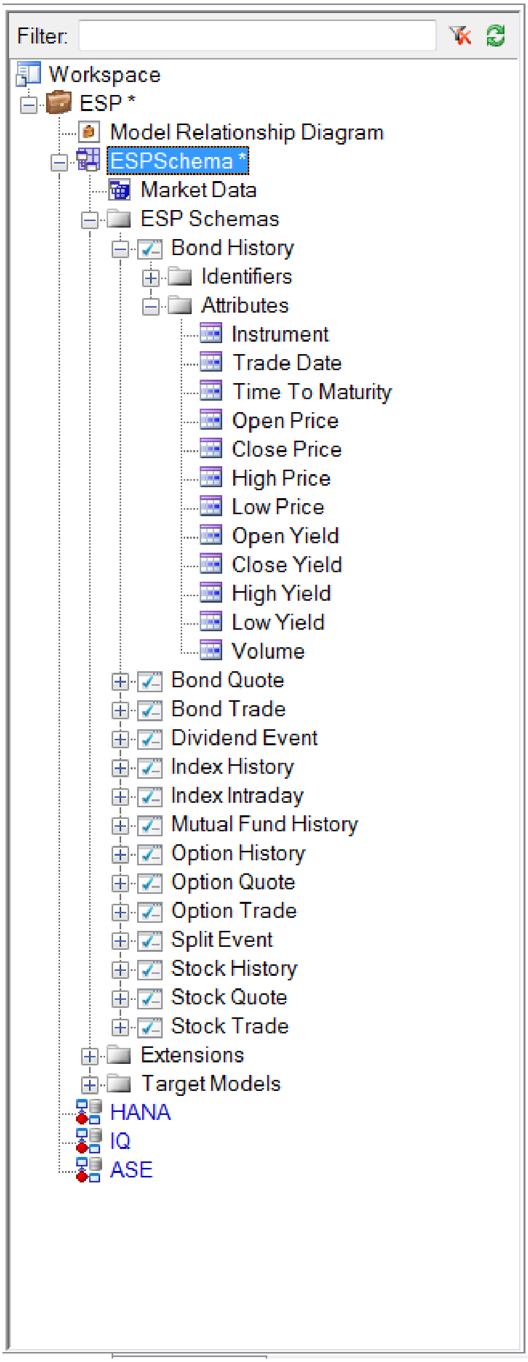
The Sample Model

The sample model includes sample ESP schema, and the SAP Sybase IQ, SAP HANA, and ASE data models.

The right frame shows the diagram selected. The top-level Model Relationship diagram shows that the ESP Schema model is related to the three physical data models by generation—that is, these physical models were generated from the logical model. You can also generate a logical model from a physical model.

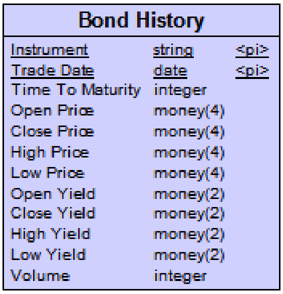
When you open the ESP Schema model and expand a schema definition and its attributes in the Browser, you see a hierarchy like this one for Bond History.

Figure 2: ESP Schema Definitions in Browser



To show the Bond History diagram, right-click it in the Browser and choose **Find in Diagram**.

Figure 3: Bond History Diagram



Bond History

Instrument	string	<pi>
Trade Date	date	<pi>
Time To Maturity	integer	
Open Price	money(4)	
Close Price	money(4)	
High Price	money(4)	
Low Price	money(4)	
Open Yield	money(2)	
Close Yield	money(2)	
High Yield	money(2)	
Low Yield	money(2)	
Volume	integer	

Creating an ESP Schema Model

Create a new ESP Schema model using the ESP Schema category, either by creating a logical model and extending it, or by generating it from an SAP Sybase IQ, SAP HANA, or ASE model that has been extended.

Creating a Model Using Categories

Use PowerDesigner to create and automatically extend any ESP Schema model type.

Prerequisites

Designate the ESP Schema set as the default category.

Task

1. Choose **File > New Model**.
2. In the New Model dialog, select **Categories**, and choose a category item:
   - **ESP Schema**
   - **SAP Sybase IQ**
   - **ASE**
   - **HANA**
3. Enter a model name.
4. Choose **OK**.

Creating a Logical Data Model

Create a logical data model and add extensions to it.

1. Choose **File > New Model**.
2. In the New Model dialog, select **Model types** and **Logical Data Model**.

3. Enter a model name.
4. Click the **Select Extensions** button to the right of the Extensions box.

   A dialog shows currently loaded extensions. You can apply extensions when you create the model or later.
5. Select **ESP Schema**, select whether to share or copy, and choose **OK**.

Option	Description
Share the extended model definitions	PowerDesigner always uses the contents of the `.xem` file. If the contents of the `.xem` file change, the model sees those changes. For example, if a future version of ESP Schema includes a new version of the file, models that share it sees those changes immediately.
Copy the extended model definitions	Copies the contents of the `.xem` file into the model. The model uses its local copy instead of the file on disk.

   With either approach, you can use other extensions besides the shipped ESP Schema extensions by creating your own `.xem` file. Although it is possible to do this by adding to the `ESPSchema.xem` file, SAP does not recommended this.

Adding Schema Definition

Add a schema definition by creating it, importing schema definitions in a CCL file, or generating it from an SAP Sybase IQ, SAP HANA, or ASE table.

Creating Schema from the Schema Definitions Container

Create a new schema definition with initial properties.

1. Open the ESP Schema model.
2. In the Browser tree, right-click the ESP Schemas container and choose **New**.
3. Complete the information in the **General** tab or other tabs.

   You can complete schema definition properties at any time before generating the physical models.
4. Click **OK** to save the schema definition.

Creating Schema with the Entity Tool

Create schema from the diagram.

1. Open the ESP Schema model.
2. In the diagram, click the **Entity** tool .

A new, empty schema definition appears in the diagram, and in the Browser tree when expanded.

3. Right-click the diagram and choose **Properties**.
4. Add attributes and identifiers in the properties sheet.

Creating a Schema from the ESP Schema Container

Create a new schema definition with initial properties.

1. Right-click the ESP Schema container and choose **New** > **ESP Schema**.
2. Complete the information in the **General** tab or other tabs.

   You can complete schema definition properties at any time before generating the physical models.
3. Click **OK** to save the schema definition.

Generating Schema from a Sybase IQ, SAP HANA, or ASE Table

Follow the same steps as when generating an ESPSchema model, selecting a single table to generate.

Defining Schema Properties

Define schema details in the properties sheet.

Prerequisites

Add the schema definition to the ESP Schema model.

Task

1. Open the ESP Schema Properties sheet from the Browser tree or the diagram.
2. Edit fields on the **General**, **Attributes**, and **Identifiers** tabs.
3. (Optional) Right-click an attribute to open the Attribute Properties sheet.
4. (Optional) In the Attribute Properties sheet, choose **More** to see extended property details.
5. Choose **Apply** to apply changes.
6. Choose **OK** when done.

General Tab Properties

View information about the Name, and Comment properties of a schema definition on the General tab of the Schema Definition Properties sheet.

Table 5. Schema Definition Properties – General Tab

Property	Description
Name	Text that identifies the object's purpose for non-technical users, for example, Stock Quote. This element is used for descriptive purposes only, and can contain any string.
Comment	An optional comment field. This is stored only in the model, not in the schema.

Attributes Tab Properties

The Attributes tab of the Schema Definition Properties sheet lets you quickly set information for all fields of an ESP Schema.

Table 6. Schema Definition Properties – Attributes Tab

Property	Description
Name	Name of the field. The value can contain any string. This element is used for descriptive purposes only.
Code	By default the code is generated from the name by applying the naming conventions specified in the model options. To decouple name-code synchronization, click to release the = button to the right of the Code field.
ESP Datatype	Select from the list of supported ESP datatypes. For information on ESP datatypes, see *Datatypes* on page 27.
Data Type (Internal PowerDesigner datatype)	Select from the list of supported datatypes. For information on PowerDesigner datatypes, see related information in *PowerDesigner>Data Modeling*.
Length	Required for money and decimal data. Limited to precision 34. Precision must be the same on SAP Sybase IQ and ASE. Not used for other datatypes.
Precision	Required for money and decimal data. Not used for other datatypes.

Property	Description
Domain	Specifies a domain which defines the datatype and related data characteristics for the schema attribute. It may also indicate check parameters, and business rules. Select a domain from the list, or click the Ellipsis button to create a new domain in the List of Domains.

Attribute Properties Sheet

Each field in a schema definition has its own Properties sheet.

In the Attribute Properties Sheet, you can:

- View or edit the same information as in the Attributes tab of the Schema Definition Properties sheet
- Specify validation checks for an attribute
- View attribute dependencies
- View impact and lineage analyses for an attribute

Adding an Attribute to Schema

Add fields to schema by adding attributes to the schema definition.

1. In the schema definition to which you are adding an attribute, do any of:
    - From the schema definition, right-click and choose **New**. This opens the Attribute Properties sheet.
    - From the Attributes tab in the ESP Schema Properties sheet, type information in the row below the last attribute.
    - From the Attributes tab in the ESP Schema Properties sheet, click one of the toolbar buttons to **Insert a Row**, **Add a Row**, or **Add Attributes** or **Replicate Attributes** from other schema definitions.

    Before replicating attributes, read *Object Replications* in *PowerDesigner Core Features Guide*.
2. Edit information in the Attributes Properties sheet or row as needed.

Identifiers

An identifier is a column or combination of columns that uniquely defines a specific ESP Schema.

Identifiers in the ESP Schema model become keys on tables in the SAP Sybase IQ, SAP HANA, and ASE physical models.

Each ESP Schema can have at most one primary identifier, which becomes the primary key in the generated table.

When an identifier has multiple attributes, the primary key in the destination table is composed of multiple columns. For example, in the sample model, the Dividend Event schema has one identifier. Attributes for this primary identifier are Instrument and Disbursed Date. Thus the primary key for the Dividend Event table is composed of both the Instrument and Disbursed Date columns.

Defining Identifiers

Define identifiers to indicate which schema attributes become keys in the destination table.

1. Either:
   - Right-click an ESP Schema and choose **New > Identifier**, or
   - (Primary identifiers only) On the ESP Schema Properties sheet, select the **Attributes** tab, and click the Primary Identifier column (the narrow column with the header **P**) for each attribute that is part of the primary identifier. Skip the remaining steps.

   Note: In the ESP Schema Properties **Attributes** tab, a checkmark in the **P** column indicates a primary identifier.
2. Select the **General** tab in the Identifier Properties sheet:
   a) (Optional) Set the identifier name.
   b) For a primary key, select **Primary Identifier**.
3. On the **Attributes** tab in the Identifier Properties sheet, enter the fields that identify the schema.

Validating a Model

Check the validity of your model after schema changes, and before generating schema templates, code, or a physical model.You can check the validity of a model at any time.

1. (Optional) Select diagrams for the schema you want to validate.
2. Choose **Tools > Check Model** (**F4**).
3. In the **Options** tab of Check Model Parameters, expand the containers and choose validation checks.

   The **Options** tab lists checks to be performed with symbols indicating their severity.
   - Do not disable any ESP-specific checks.
   - (Default and recommended) Disable **Existence of relationship or entity link** under **Entity**.
4. In the **Selection** tab, navigate to the ESP Schemas subtab and select schema definitions to check:
   - Select or unselect check boxes.
   - Choose a named selection.

- If you selected schema in your diagram before starting the model check, you can select them for checking by clicking **Use Graphical Selection** () in the Selection tab toolbar.

5. Choose **OK**.

Next

Review results in the **Check Model** subtab in the status bar. It lists the checks made, and any errors or warnings.

Correct any errors. No automatic corrections are provided.

PowerDesigner Validity Checks

Standard PowerDesigner checks determine if a model is internally consistent and correct.

For example:

- Each ESP Schema name must be unique
- Each object name in an ESP Schema model must be unique
- Each field must have an assigned ESP Data Type.

For descriptions of standard PowerDesigner checks, see *Working with Data Models > Checking a Data Model* in the PowerDesigner *Data Modeling* guide.

Custom Checks for ESP Schema Extensions

The ESP Schema extension offers many custom checks.

Checks for Each Schema

Custom checks under Schema Definition type validate values in the General tab of the Schema Properties sheet.

Table 7. Schema Definition Custom Checks

Option	Validates
NameIsValid	Names of ESP Schema must be valid java identifiers.

Checks for Each Field in a Schema

Custom checks for fields are under Entity Attribute type. They validate values in the Attributes tab of the ESP Schema Properties sheet.

Table 8. Attribute Custom Checks

Option	Validates
FieldNameIsValid	Field names must be valid java identifiers.

Option	Validates
ESPDatatypeExists	Datatype is specified
UniqueDestColumnName	DestColumnName is unique within that schema

Importing a CCL File

Import the defined schema definitions in a CCL file into an ESP Schema model.

1. Open the ESP Schema model.
2. In the **Browser** tree, right-click the ESP Schema container and choose **Import CCL File...**.
3. Navigate to the CCL file you wish to import.
4. Click **OK** to import the schema definitions defined in the CCL file.

Note: A warning message appears if the CCL file schema definitions are not valid. You must resolve the errors before importing the CCL file. Navigate to the `User/Appdata/Roaming/PowerDesigner/EspCompiler/compiledOutput.log` file to view the errors.

The schema defined in the CCL file is imported into the ESP Schema model.

Exporting a CCL File

Export all the defined schema from the ESP Schema model into a CCL file for compiling and further analysis.

1. Open the ESP Schema model.
2. In the **Browser** tree, right-click the ESP Schema container and choose **Export CCL File...**.
3. Navigate to the CCL file you wish to export to.
4. Click **OK** to export the schema definitions.

The schema defined in the ESP Schema model is exported as a CCL file.

Model Generation

Model generation with the ESP Schema models is a critical step in ensuring the integrity of your production environment.

You can either:

- Generate SAP Sybase IQ, SAP HANA, and ASE physical data models from an ESP Schema model, or

- Generate an ESP Schema logical data model from an SAP Sybase IQ, SAP HANA, or ASE physical data model

Each generation process relies on transformation rules for that model type, which are defined in the ESP Schema extensions for PowerDesigner.

Generating a new Sybase IQ, SAP HANA, or ASE Model from an ESP Schema Model

Generate either an SAP Sybase IQ, SAP HANA, or an SAP Sybase ASE physical data model from an ESP Schema logical data model.

1. Open the ESPSchema model.
2. From the PowerDesigner main menu, choose **Tools > Generate Physical Data Model**.
3. In the **General** tab of the PDM Generation Options dialog, choose **Generate new Physical Data Model**.
4. For a new model, choose the target DBMS and the appropriate Name and Code.

   - For HANA, choose:

Field	Value
DBMS	**SAP HANA Database 1.0**
Name	Keep the default, ESPSchema, or enter another name.
Code	Auto-generated from Name. For example, when Name is ESPSchema, Code is ESPSCHEMA.

   - For ASE, choose:

Field	Value
DBMS	**SAP Sybase Adaptive Server Enterprise 15.7**
Name	Keep the default, ESPSchema_1 (the name of the container), or enter another name.
Code	Auto-generated from Name. For example, when Name is ESPSchema_1, Code is ESPSCHEMA_1.

   - For IQ, choose:

Field	Value
DBMS	**SAP Sybase IQ 15.X** **Note:** Use latest version of SAP Sybase IQ available.
Name	Keep the default, ESPSchema_1 (the name of the container), or enter another name.

Field	Value
Code	Auto-generated from Name. For example, when Name is ESPSchema_1, Code is ESPSCHEMA_1.

5. Click the **Detail** tab.
6. (Optional) Choose **Check model** and **Save generation dependencies**.
7. Ensure that **Enable transformations** is selected.
8. Click the **Extensions** tab and ensure that the appropriate extension is selected:
    - **ASE** when generating a new SAP Sybase ASE model
    - **IQ** when generating a new SAP Sybase IQ model
    - **HANA** when generating a new SAP HANA model
9. On the **Pre-generation** and **Post-generation** tabs, ensure that all transformation rules are selected.

    The post-generation tab appears only for new models.
10. On the **Selection** tab, select **ESPSchema** to create tables for SAP Sybase IQ, SAP HANA, or SAP Sybase ASE, and choose **OK**.

Next

After generation, check indexes, set physical options, and add foreign keys as needed.

Checking Indexes

PowerDesigner creates default indexes. Add, edit, or remove them as needed.

1. Open the new or updated physical data model.
2. For each table, right-click the table and choose **Properties**.
3. In the **Indexes** tab, edit indexes as needed for your data and performance requirements.

Setting Physical Options

Set physical options for each table as needed for your SAP Sybase IQ or SAP Sybase ASE database.

1. Right-click the table and choose **Properties**.
2. Define any options needed.

    - (SAP Sybase ASE only) In the **Physical Options (Common)** tab, choose from the physical options most commonly set for the object.
    - In the **Physical Options** tab, choose from all available options.
    - (SAP Sybase ASE only) In the **Partitions** tab, set partitioning options for selected columns.

    For more information on partitioning, see the SAP Sybase ASE and SAP Sybase IQ documentation sets.

Adding Foreign Keys

Add foreign-key relationships to physical data models.

1. Add tables to the physical data model that are not in your Market Data diagram and that contain lookup columns for foreign keys.

   New ASE, SAP HANA, and SAP Sybase IQ models generated from a ESP Schema model contain only market data tables.
2. Right-click the table and choose **Properties** or **Keys**.
3. Add foreign-key relationships to tables that are not in the Market Data diagram.

Generating a new ESP Schema Model from a Sybase IQ, SAP HANA, or ASE Model

Generate a new ESP Schema logical data model from either SAP Sybase IQ, SAP HANA, or ASE physical data models.

1. Open either the **IQ**, **HANA**, or **ASE** model.
2. From the PowerDesigner main menu, choose **Tools** > **Generate Logical Data Model**.
3. In the **General** tab of the LDM Generation Options dialog, choose **Generate new Logical Data Model**.
4. Specify a **Name**.

   Code is autogenerated from the name.
5. On the **Detail** tab, choose Options:
   - (Optional) Check model
   - (Optional) Save generation dependencies
   - (Optional) Convert names into codes
   - (Required) Enable transformations
6. On the **Extensions** tab, choose **ESPSchema**.
7. On the **Selection** tab, choose tables from which to generate schema.
8. Choose **OK**.

Updating an existing Sybase IQ, SAP HANA, or ASE Model from an ESP Schema Model

Update either an SAP Sybase IQ, SAP HANA, or a ASE physical data model from a ESP Schema logical data model.

1. Open the ESP Schema model.
2. From the PowerDesigner main menu, choose **Tools** > **Generate Physical Data Model**.

3. In the **General** tab of the PDM Generation Options dialog, choose **Update existing Physical Data Model**.
4. Select the model and leave **Preserve Modifications** selected.
5. Click the **Detail** tab.
6. (Optional) Choose **Check model** and **Save generation dependencies**.
7. Ensure that **Enable transformations** is selected.
8. In the Merge Models dialog, confirm the updates you want and choose **OK**.

Next

After generation, check indexes, set physical options, and add foreign keys as needed.

Updating an existing ESP Schema Model from a Sybase IQ, SAP HANA, or ASE Model

Update an existing ESP Schema logical data model from either SAP Sybase IQ, SAP HANA, or ASE physical data models.

1. Open either the **IQ**, **HANA**, or **ASE** model.
2. From the PowerDesigner main menu, choose **Tools > Generate Logical Data Model**.
3. In the **General** tab of the LDM Generation Options dialog, choose **Update existing Logical Data Model**.
4. Select the model and leave **Preserve Modifications** selected.
5. On the **Detail** tab, choose Options:
   - (Optional) Check model
   - (Optional) Save generation dependencies
   - (Optional) Convert names into codes
   - (Required) Enable transformations
6. On the **Selection** tab, choose tables from which to generate schema.
7. Choose **OK**.

Impact and Lineage Analysis

With impact and lineage analysis, you can determine the full impact of changes to any object in the integrated model.

Impact analysis shows the effect of an action on the objects that depend on the initial object.

Lineage analysis identifies the objects that influence the initial object.

You can perform these analyses on:

- A schema definition or any of its properties in the ESP Schema logical data model

- A table or column in the ASE, SAP HANA, or SAP Sybase IQ physical data model

The results shows the effect of a change throughout the logical and physical data models.

Launching an Impact and Lineage Analysis

Analyze the impact of a change to your model from the Impact and Lineage Analysis dialog box.

The Impact and Lineage Analysis dialog lets you review your analysis through:

- A preview – displays the impact and lineage analysis in a tree form (see *PowerDesigner Core Features Guide > Reviewing an Analysis in Preview*).
- An impact analysis model (IAM) – displays the impact and lineage analysis in a diagram (see *PowerDesigner Core Features Guide > Reviewing an Analysis in an IAM Model*).

1. Open an impact and lineage analysis in any of these ways:

- Select an object in the Browser or in the diagram and press **Ctrl** + **F11**.
- Select one or more objects in the diagram and select **Tools > Impact and Lineage Analysis**.
- Right-click an object symbol in the diagram and select **Edit > Impact and Lineage Analysis**.
- Right-click an object entry in the Browser and select **Impact and Lineage Analysis**.
- (When deleting an object) Click **Impact** on the Confirm Deletion dialog box.
- Open an object's property sheet, click the **Dependencies** tab, then click **Impact Analysis** .

2. (Optional) Enter a name for your analysis result. This becomes the name of the generated model.

3. Select an impact rule set for your analysis. Choose one of these predefined rule sets:

- Conceptual Impact Analysis – restrict the analysis to objects impacted by modeling changes on the initial object, such as a modification on a requirement definition.
- Data Impact Analysis – identify the use, if any, of a value contained in the initial object.
- Delete Impact Analysis – (default when deleting an object) restrict the analysis to objects that are directly impacted by the deletion of the initial object.
- Global Impact Analysis – (default when not deleting an object) identify all the objects that depend on the initial object.
- None – no impact rule set is selected.

4. Select a lineage rule set for your analysis. Choose one of these predefined rule sets:

- Conceptual Lineage Analysis – justify the modeling existence of the initial object, and ensure it fulfills a well-identified need.
- Data Lineage Analysis – identify the origin of the value contained in the initial object.

- Global Lineage Analysis – (default when not deleting an object) identify all the objects that influence the initial object.
- None – (default when deleting an object) no lineage rule set is selected.

5. (Optional) Click the **Properties** tool next to each rule set to review it (see *PowerDesigner Core Features Guide > Editing analysis rules*).

   The analysis appears in the **Impact and Lineage** tab of the dialog box (see *PowerDesigner Core Features Guide > Reviewing an Analysis in Preview*).

Note: You can click the **Select Path** tool to change the default folder for analysis rule sets, or click the **List of Rule Sets** tool to open the **List of Impact and Lineage Analysis Rule Sets** window, and review a specific rule.

Generating an Analysis Diagram

Generate an analysis diagram to view the impact or lineage analysis in graphical form.

Prerequisites

Launch an impact or lineage analysis.

Task

1. In the Impact and Lineage Analysis dialog, click **Generate Diagram** to view a graphical form of the analysis in its default diagram.
2. (Optional) Save (**Ctrl+S**) the diagram as an impact analysis model (IAM).

   See *PowerDesigner Core Features Guide > Reviewing an Analysis in an IAM Model*.

Reviewing an Impact and Lineage Analysis

Review the analysis in the preview or the impact and lineage model diagram.

1. Review the impact of the action and the lineage of the entity in the preview.
2. In the preview **List** tab, save the analysis in RTF or CSV format, or print.
3. You can refine your analysis by removing or adding initial objects, changing the analysis rule sets to be used, and customizing actions.
4. If you have generated an IAM, you can customize the display preferences and model options, print the model, and compare it with another IAM.
5. Watch for a red dot on an object icon in a generated model.

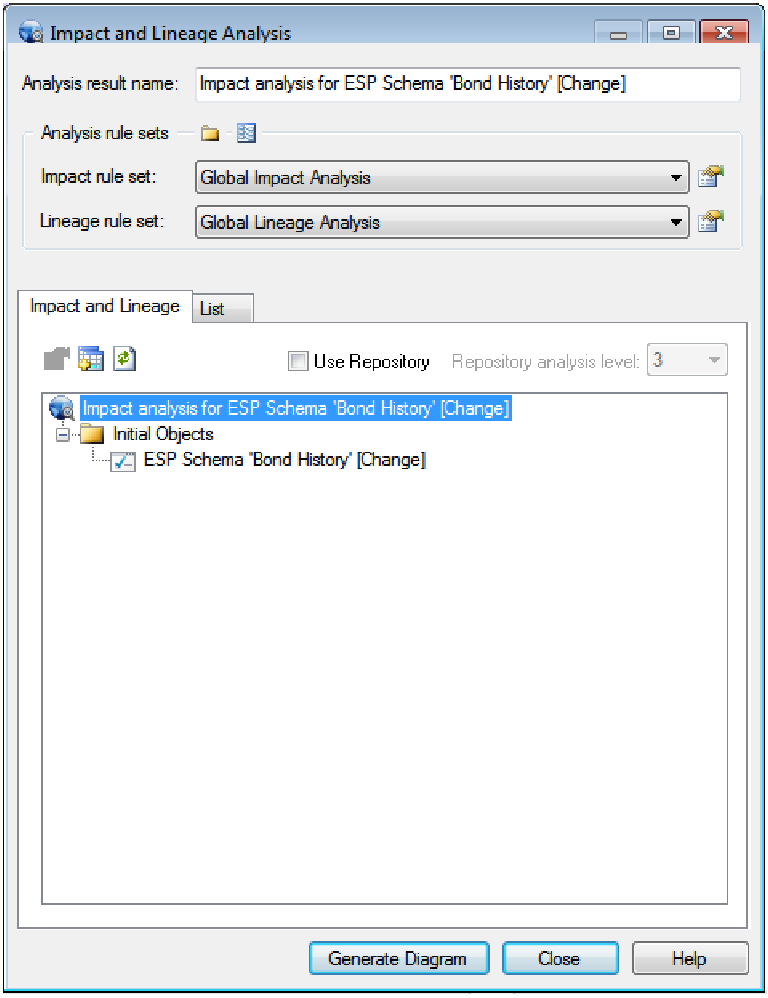
   When you generate a model to another model or create an external shortcut, you create cross-model dependencies, which are taken into account during impact and lineage analysis.

   When an object belonging to an unavailable related model is encountered, a red dot appears on the object icon and the analysis is interrupted. To continue, open the related model by right-clicking the object in the IAM Browser or in the preview, and selecting **Open Model**.

Sample Analysis for a Schema Definition

The sample analysis for a schema definition shows that the Bond History schema in the ESP Schema model was used to generate the `BOND_HISTORY` tables in the SAP HANA, ASE, and SAP Sybase IQ models.

Figure 4: Impact Analysis Example for Schema Definition



Impact and Lineage Analysis

Analysis result name: Impact analysis for ESP Schema 'Bond History' [Change]

Analysis rule sets

Impact rule set: Global Impact Analysis

Lineage rule set: Global Lineage Analysis

□ Use Repository   Repository analysis level: 3

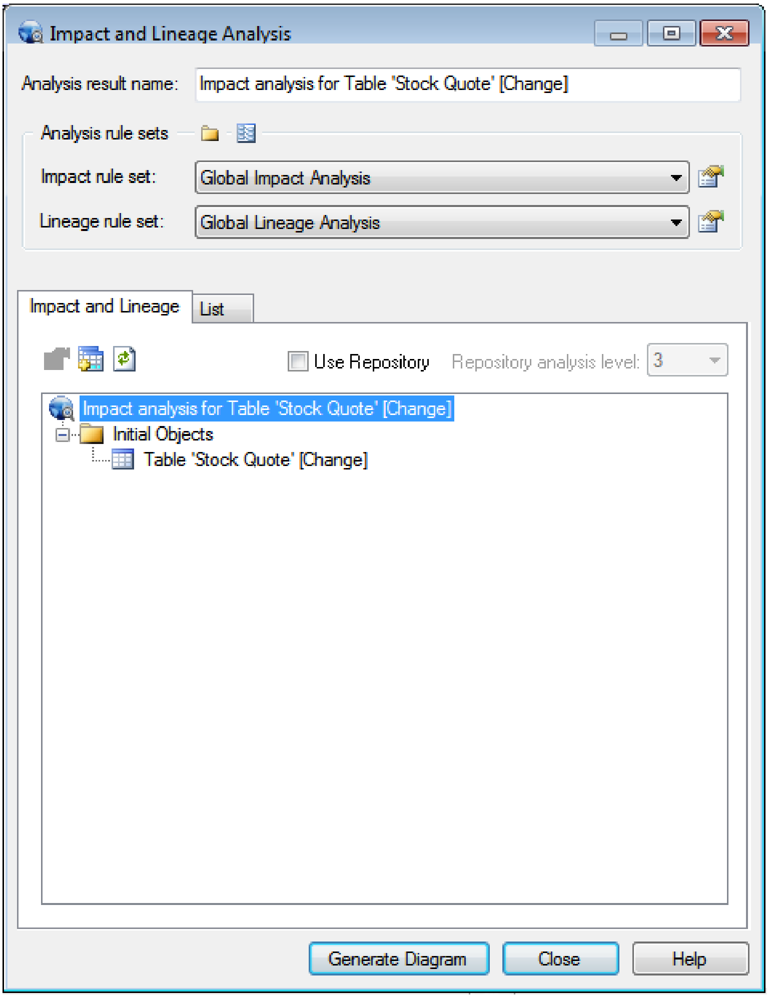
- Impact analysis for ESP Schema 'Bond History' [Change]
  - Initial Objects
    - ESP Schema 'Bond History' [Change]

Sample Analysis for a Table

The sample analysis for a table shows that the STOCK_QUOTE table was generated from the Stock Quote schema definition in the ESP Schema model.

Outgoing References shows foreign-key relationships. ESP Schema definitions become Market Data diagram tables when generated to a PDM.

Figure 5: Impact and Lineage Analysis for STOCK_QUOTE Table in ASE



Impact and Lineage Analysis

Analysis result name: Impact analysis for Table 'Stock Quote' [Change]

Analysis rule sets

Impact rule set: Global Impact Analysis

Lineage rule set: Global Lineage Analysis

□ Use Repository   Repository analysis level: 3

Impact analysis for Table 'Stock Quote' [Change]
- Initial Objects
  - Table 'Stock Quote' [Change]

DDL Script Generation

The data models for the SAP Sybase IQ, SAP HANA, and ASE databases target different databases; however, they share an almost identical structure. Modify data models by creating additional tables or columns to suit your business environment.

The corresponding adapter (IQ, HANA, ASE) schema must match the SAP Sybase IQ, HANA, and ASE database schema for all tables in which data is inserted. After you make changes, you can use PowerDesigner to produce a set of data definition language (DDL) statements directly from the physical data model (IQ, HANA, and ASE). PowerDesigner saves the DDL statements in a SQL script that you can run to generate the tables and other objects for the target databases.

Generating Database Schema with PowerDesigner

PowerDesigner includes all the resources you need to generate a set of DDL statements in SQL scripts directly from the PowerDesigner data models. Run these scripts to generate a schema for your SAP Sybase IQ, SAP HANA, and ASE databases.

1. In PowerDesigner, open the data model.
2. Change the default database user.
3. Generate the script that creates a schema for the new database.
4. Log in to the database and run the script.

Changing the Default Database User

Overwrite the default database owner for the SAP Sybase IQ, SAP HANA, or ASE database with a name specific to your environment.

In the database, the user who creates an object (table, view, stored procedure, and so on) owns that object and is automatically granted all permissions on it. Overwriting the default user name globally changes ownership of database objects from the default owner to the new owner.

1. Start PowerDesigner.
2. Select **File** > **Open** and choose the database that you want to change the default owner of (`IQ.pdm`, `HANA.pdm`, or `ASE.pdm`).
3. Select **Model** > **Users and Roles** > **Users.**
4. In the Name and Code columns, change the default user to the new database user.
5. Click **OK**.

Generating DDL Scripts

Generate DDL scripts directly from the SAP Sybase IQ, SAP HANA, or SAP Sybase ASE data model. PowerDesigner saves the results in a SQL script that you can use to generate the tables and other objects in the target database.

Use the model file of the database for which you wish to generate DDL scripts. For example, to generate DDL for the SAP Sybase ASE database, use the ASE model. When you have the model open, do not change the target database as doing so results in the loss of index information.

By default, the `ASE.pdm` data model includes only those indexes that support the sample queries. The statements that create these indexes are included in the DDL scripts, which means the indexes supplied with the model are created automatically when you run the corresponding DDL scripts.

You can add or remove indexes from the ASE data model. For detailed information on SAP Sybase ASE indexes, see the SAP Sybase ASE product documentation.

1. Select **Database > Generate Database**.
2. Browse to the directory where you want to store the script. Click **OK**.
3. Enter a name for the SQL script.
4. On the Options tab, verify that the options are set correctly:

Object	Options
Domain	Create User-Defined Data Type
Table	Create Table
Column	User Data Type
Key	Create Primary Key Inside
Index	• Create Index • Index Filter Foreign Key • Index Filter Alternate Key • Index Filter Cluster • Index Filter Others

5. Click the **Selection** tab.
6. Choose the database owner.
7. On the **Tables** tab, click **Select All**.
8. On the **Domains** tab, choose the database owner, click **Select All**, click **Apply**, then click **OK**.

PowerDesigner checks the model for any errors, builds a result list, and generates the DDL. The Result dialog appears, which identifies the name and location of the generated file.

9. You can click **Edit** to view the generated script.

   The Result List dialog appears in the background and may include several warnings, for example, `"Existence of index"` and `"Existence of reference".` You can safely ignore these warnings.

10. Close the Result List dialog, then exit PowerDesigner.

- If PowerDesigner prompts you to save the current workspace, click **No**.
- If PowerDesigner prompts you to save the model, click **Yes** to save the modified model. Otherwise, click **No**.

Executing DDL Scripts for the SAP Sybase IQ Database

Execute the DDL script in Interactive SQL and create database objects in the SAP Sybase IQ database.

Prerequisites

Start the SAP Sybase IQ database server if it is not running.

Task

1. In a command prompt, change to the directory that contains the database files and enter:

   ```
   start_iq -n server_name @config_file.cfg database_name.db.
   ```

   Use the **-n** switch to name the server, either in the configuration file or on the command line when you start the server.

   Note: If you specify **-n** *server_name* without a *database_name*, you connect to the default database on the current server. If you specify **-n** *database_name* without a *server_name*, you connect to the specified database on the current server.

2. Enter **dbisql.**
3. Enter the correct user ID, password, and server information.
4. Open the generated DDL script for SAP Sybase IQ and click **Execute SQL Statement** on the toolbar.

Executing DDL Scripts for the SAP HANA Database

Execute the DDL script using hdbsql and create database objects in the SAP HANA database.

Prerequisites

Start the SAP HANA database server if it is not running.

Task

In a command prompt, enter:

```
hdbsql -n <host>:<port> -u <user> -p <password> -I <script file>
```

Executing DDL Scripts for the ASE Database

Execute the DDL script in Interactive SQL and create database objects in the ASE database.

Prerequisites

Start the ASE server if it is not running.

Task

1. In a command prompt, enter:

   ```
   isql -S <server_name> -U <user_name> -P <password> -i <ase_ddl.sql
   file> -o <log_file>
   ```
2. PowerDesigner prompts:
   - To save the current workspace. Click **No**
   - To save the (modified) model. Click **Yes** or **No**.
3. Check the log file for errors.

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

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
xmlattributes	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/Monticello	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-Saskatchewan	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 %.

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 C Performance and Tuning Tips

Optimizing performance in SAP Sybase Event Stream Processor requires tuning at the project level as well as at the infrastructure level (machine, OS, network configuration, and so on).

If you tune your projects to produce maximum throughput and minimum latency but do not configure your infrastructure to handle the throughput, you will see sub-optimal performance. Likewise, if you configure your infrastructure to handle maximum throughput but do not tune your projects, your performance suffers.

Distributing Load through Parallelization

To improve performance of large ESP projects, separate the data into smaller chunks that are processed within their own partitions. Processing on multiple partitions in parallel can improve performance over processing in one large partition.

There are various ways to parallelize an ESP project.

1. Application-based Partitioning

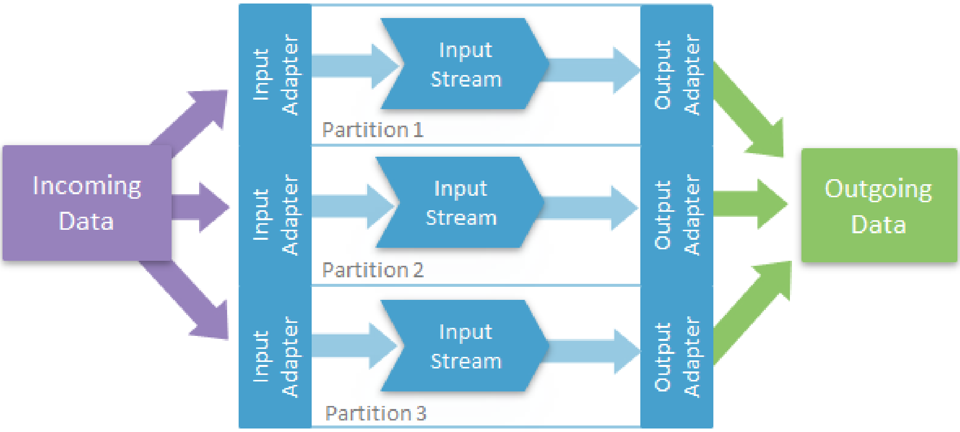
You can send all incoming data to each of the input adapters within your ESP project, and then attach each of these adapters to a stream or delta stream that filters a subset of the overall incoming data. The output adapters receive this data and output it to the external datasource.

Advantages:

- You can improve performance and process high volumes of data since having multiple streams processing subsets of the data divides the load on the processor.
- You also have the advantage of not having to create a custom adapter or do any custom coding aside from specifying the filtering.
- Can partition across cores, but is best suited for partitioning across machines.

Disadvantages over other methods:

- You have to duplicate the input data feeding into the input adapters.



2. Partitioning Using a Custom Adapter

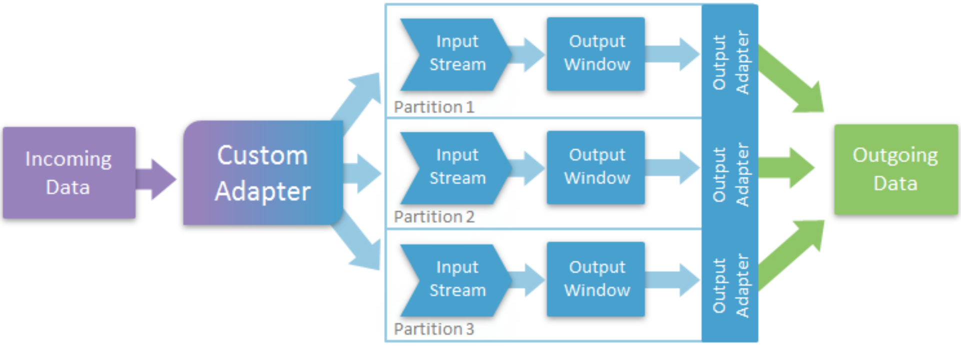
You can write a custom adapter to receive input data and publish it to various streams, delta streams, or windows on separate machines. These streams or windows would then process and send this data to separate output adapters which would then publish it to the end datasource. The custom adapter is responsible for partitioning the input data in this scenario.

Advantages:

- You can improve performance and process high volumes of data by filtering incoming data across multiple machines.
- You can customize your adapter to meet your partitioning requirements.
- You do not need to duplicate any data.
- Can partition across cores, but is best suited for partitioning across machines.

Disadvantage over other methods:

- Requires more effort in terms of coding because you have to write a custom adapter as you cannot currently partition the available adapters provided with Event Stream Processor.

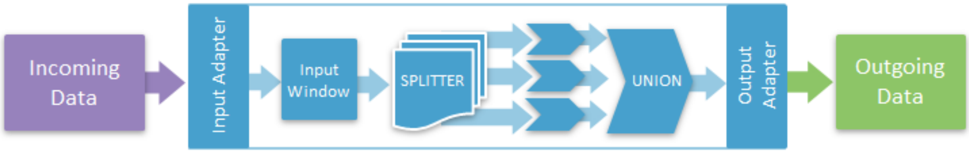


3. Partitioning Using a SPLITTER Statement

You can use the CCL SPLITTER object to subdivide input data based on specific criteria, and then a UNION statement to consolidate the data before sending it to the output adapter.

Advantages:

- You have more flexibility in terms of the operations that you can perform on the streams resulting from the SPLITTER. For example, you first split the data, perform operations on the resulting streams, and then consolidate the data again.
- Can partition across cores.



Although the example in the illustration uses a single input adapter, you can use a SPLITTER when using multiple input adapters.

Note: Using the JOIN object does not realize the same performance benefit as using the UNION. In fact, the JOIN operation can degrade performance considerably, so to optimize performance, parallelizing your project using the SPLITTER/UNION combination is recommended over using JOIN.

In both the cases, the number of parallel instances is limited to the throughput of the union and, when used, the SPLITTER. In addition, the number of parallel instances depends on the number of available CPUs.

4. Automatic Partitioning

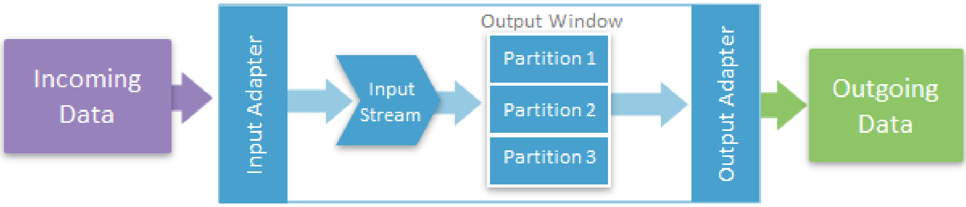
You can create multiple parallel instances of a given element (delta stream, stream, window, module) and partition input data into these instance. Partitioning data this way results in higher performance as the workload is split across the parallel instances. If using this scenario, you can partition the CCL elements using CUSTOM, HASH, or ROUND ROBIN partitioning.

Advantages:

- Ideal for complex projects which perform computationally expensive operations, such as aggregation and joins.
- Quick and easy to add this functionality to your project.
- Can partition across cores.

Disadvantage:

- Lacking the ability of ordering the output.



General Guidelines

Hash partitioning uses hash functions to partition data. The hash function determines which partition to place a row into based on the column names you specify as keys. These do not have to be primary keys. Round-robin partitioning distributes data evenly across partitions without any regard to the values.

Choose a type based on the calculations you are performing on the input data. For example, round-robin is sufficient for stateless operations like simple filters, but not for aggregation as this would produce differing results. Hash partitioning is necessary for grouping records together, but grouping may not evenly distribute the data across instances.

When implementing the scenarios above, you can use round-robin or key-based partitioning. Round-robin partitioning provides the most even distribution across the multiple parallel instances, but is recommended only for projects limited to insert operations (that is, no updates or deletes). For projects using insert, update, and delete operations, key-based partitioning is preferable. Any update or delete operation on a record should occur on the same path where the record was inserted, and only key-based partitioning can guarantee this. However, key-based partitioning can distribute load unevenly if the HASH function is not applied correctly, which results in some partitions with a higher burden than others.

For an example of improving performance throughput using automatic partitioning, see *Performance Tuning Using Partitioning* in the *Examples Guide*.

For more information on the SPLITTER and UNION statements, see the *Programmers Reference Guide* and refer to the `splitter`, `Union`, and `RAP_splitter_examples` provided in your `Examples` folder.

Distributing Load through Modularization

You can optimize performance by breaking projects into modules. This strategy spreads the load out to more cores, thereby increasing throughput.

Use modules to double, quadruple, and so on, the number of partitions, with very little additional code. The more partitions you create, the more you distribute the load.

For information on modularity, see the *Programmers Reference*, the Continuous Computation Language chapter in the *Getting Started Guide*, and the `Submodules` example provided in your `examples` folder.

Data Flow in Event Stream Processor

The throughput of an Event Stream Processor project depends on the throughput of the slowest component in the project.

Each stream in ESP has an internal queue that holds up to 1024 messages. This queue size is hard-coded and cannot be modified. An internal queue buffers data feeding a stream if that stream is unable to keep up with the inflowing data.

Consider an example where data flows from an input adapter, through streams A, B, and C, and then through an output adapter. If the destination target of the output adapter cannot handle the volume or frequency of messages being sent by the output adapter, the internal queue for the stream feeding the output destination fills up and stream C cannot publish additional messages to it. As a result, the internal queue for stream C also fills up and stream B can no longer publish to it.

This continues up the chain until the input adapter can no longer publish messages to stream A. If, in the same example, the input adapter is slower than the other streams, messages will continue being published from stream to stream, but the throughput is constrained by the speed of the input adapter.

Note that if your output destination is a database, you can batch the data for faster inserts and updates. Set the batch size for a database adapter in the `service.xml` file for the database. For information on configuring the `service.xml` file, see the *Configuration and Administration Guide*.

Batching data carries some risk of data loss because the database adapters run on an in-memory system. To minimize the risk of data loss, set the batch size to 1.

Log Store Considerations

The size and location of your log stores can impact performance.

Sizing the log stores correctly is important. A store that is too small requires more frequent cleaning cycles, which severely degrades performance. In the worst case, the log store can overflow and cause the processing to stop. A store that is too large also causes performance issues due to the larger memory and disk footprint. For detailed information on calculating the optimal log store size, see *Basic Administrative Tasks > Sizing the Log Store* in the *Configuration and Administration Guide*.

When storing ESP data locally using log stores, use a high-speed storage device (for example, a raid array or SAN, preferably with a large dynamic RAM cache). Putting the backing files for log stores on single disk drives (whether SAS, SCSI, IDE, or SATA) always yields moderately low throughput.

Note: On Solaris, putting log files in `/tmp` uses main memory.

Batch Processing

When stream processing logic is relatively light, inter-stream communication can become a bottleneck. To avoid such bottlenecks, you can publish data to the ESP server in micro batches. Batching reduces the overhead of inter-stream communication and thus increases throughput at the expense of increased latency.

ESP supports two modes of batching: envelopes and transactions.

- **Envelopes –** When you publish data to the server using the envelope option, the server sends the complete block of records to the source stream. The source stream processes the complete block of records before forwarding the ensuing results to the dependent streams in the graph, which in turn process all the records before forwarding them to their dependent streams. In envelope mode, each record in the envelope is treated atomically so a failure in one record does not impact the processing of the other records in the block.
- **Transactions –** When you publish data to the server using the transaction option, processing is similar to envelope mode in that the source stream processes all of the records in the transaction block before forwarding the results to its dependent streams in the data graph. Transaction mode is more efficient than envelope mode but there are some important semantic differences between the two.

  The key difference between envelopes and transactions is that in transaction mode, if one record in the transaction block fails, then all records in the transaction block are rejected and none of the computed results are forwarded downstream.

  Another difference is that in transaction mode, all resultant rows produced by a stream, regardless of which row in the transaction block produced them, are coalesced on the key field. Consequently, the number of resulting rows may be somewhat unexpected.

In both the cases the number of records to place in a micro batch depends on the nature of the model and needs to be evaluated by trial and error. Typically, the best performance is achieved when using a few tens of rows per batch to a few thousand rows per batch. Note that while increasing the number of rows per batch may increase throughput, it also increases latency.

Main Memory Usage

There are no SAP Sybase Event Stream Processor configuration settings that directly set up or control RAM usage on the machine. However, ESP reference counts records in the system, ensuring that at most one copy of a record is present in memory, although multiple references to that record may exist in different streams.

Memory usage is directly proportional to the number of records in a project. To limit the amount of memory the entire instance of ESP uses before it reports an out-of-memory condition, use the **ulimit** command to restrict the amount of memory available to each shell process.

Determining Stream Memory Usage

When the server is running at log level 7 and it is shutdown cleanly, it reports the amount of memory consumed by every stream and any aggregation indices in the server log file.

The log level is a project configuration option you can set on the **Advanced** tab of the Project Configuration editor in Studio.

You do not have to constantly run the server at log level 7 to print memory usage statistics; the statistics are printed as long as the level is set at 7 when the server is shutting down. To change the log level at run time, use the **esp_client** tool and execute:

```
esp_client -p [<host>:]<port></workspace-name/project-name> -c
<username>:<password> "loglevel 7"
```

Not all of the memory consumed by the server is reported when the server shuts down. Therefore, the total of the reported memory will not be equal to the memory reported by system utilities (Task Manager in Windows or top in Linux) for the ESP Server. Components that potentially consume significant amounts of memory that are not reported include:

- Input queues for streams.
- SPLASH data structures such as vectors and dictionaries used.
- Indices required for handling Retention and Aging.
- Subscriber queues.
- Memory required to process SQL subscriptions that require aggregation and pulsed subscriptions.

The following sample illustrates the memory usage statistics reported in the log file:

```
[SP-6-131039] (189.139) sp(21115) CompiledSourceStream(W1):
Collecting statistics (this could take awhile).
[SP-6-131040] (190.269) sp(21115) CompiledSourceStream(W1): Memory
usage: 1,329,000,000 bytes in 3,000,000 records.
[SP-6-114012] (190.269) sp(21115) Platform(cepqplinux1)::run() --
cleaning up CompiledAggregateStream(grpbyout).
[SP-6-131039] (191.065) sp(21115) CompiledAggregateStream(grpbyout):
Collecting statistics (this could take awhile).
[SP-6-124001] (191.065) sp(21115)
CompiledAggregateStream(grpbyout)::Memory usage: 1,545,000,000 bytes
in aggregation index.
[SP-6-131039] (195.957) sp(21115) CompiledAggregateStream(grpbyout):
Collecting statistics (this could take awhile).
[SP-6-131040] (196.267) sp(21115) CompiledAggregateStream(grpbyout):
Memory usage: 1,020,000,000 bytes in 3,000,000 records.
[SP-6-114012] (196.267) sp(21115) Platform(cepqplinux1)::run() --
cleaning up CompiledAggregateStream(grpbyout2).
[SP-6-131039] (197.038) sp(21115)
CompiledAggregateStream(grpbyout2): Collecting statistics (this
could take awhile).
[SP-6-124001] (197.039) sp(21115)
CompiledAggregateStream(grpbyout2)::Memory usage: 1,545,000,000
bytes in aggregation index.
[SP-6-131039] (202.184) sp(21115)
CompiledAggregateStream(grpbyout2): Collecting statistics (this
could take awhile).
[SP-6-131040] (202.496) sp(21115)
CompiledAggregateStream(grpbyout2): Memory usage: 1,122,000,000
bytes in 3,000,000 records.
[SP-6-114012] (202.496) sp(21115) Platform(cepqplinux1)::run() --
cleaning up CompiledStream(coutputwin).
[SP-6-131039] (202.496) sp(21115) CompiledStream(coutputwin):
Collecting statistics (this could take awhile).
[SP-6-131040] (203.654) sp(21115) CompiledStream(coutputwin): Memory
usage: 651,000,000 bytes in 3,000,000 records.
```

CPU Usage

SAP Sybase Event Stream Processor automatically distributes its processing load across all the available CPUs on the machine. If the processing of a data stream seems slow, monitor each stream's CPU utilization using either the **esp_monitor** utility from the command line or through Sybase Control Center. If the monitoring tool shows one stream in the project using the CPU more than other streams, refine the project to ensure that the CPU is used evenly across the streams.

In addition to the CPU usage per stream as reported by the monitoring tools, the queue depth is also a very important metric to monitor. Each stream is preceded by a queue of input records. All input to a given stream is placed in the input queue. If the stream processing logic cannot process the records as quickly as they arrive to the input queue, the input queue can grow to a maximum size of 1,024 records. At that point, the queue stops accepting new records, which results in the automatic throttling of input streams. Since throttled streams require no CPU time, all CPU resources are distributed to the streams with the full queues, in effect performing a CPU resource load balance of the running project. When a stream's input queue is blocked, but the stream has managed to clear half of the pending records, the queue is unblocked, and input streams can proceed to supply the stream with more data.

If this inherent load balancing is insufficient to clear the input queue for any given stream, the backup of the queue can percolate upward causing blockages all the way up the dependency graph to the source stream. If your monitoring indicates growing or full queues on any stream or arc of streams in the directed graph, examine this collection of streams to determine the cause of the slow processing.

TCP Buffer and Window Sizes

High throughput data transfers between clients and SAP Sybase Event Stream Processor rely on the proper tuning of the underlying operating system's TCP networking system.

The data generated by clients for delivery to ESP does not always arrive at a uniform rate. Sometimes the delivery of data is bursty. In order to accommodate large bursts of network data, large TCP buffers, and TCP send/receive windows are useful. They allow a certain amount of elasticity, so the operating system can temporarily handle the burst of data by quickly placing it in a buffer, before handing it off to ESP for consumption.

If the TCP buffers are undersized, the client may see TCP blockages due to the advertised TCP window size going to zero as the TCP buffers on the ESP server fill up. To avoid this scenario, tune the TCP buffers and window sizes on the server on which ESP is running to between one and two times the maximum size that is in use on all client servers sending data to ESP.

For information and best practices for determining and setting TCP buffer and window sizes, consult the documentation provided with your operating system.

Improving Aggregation Performance

Aggregation functions typically require the server to iterate over every element in a group. For this reason, the performance of the aggregation operator is inversely proportional to the size of the group.

Aggregation functions can be used in a SELECT statement along with a GROUP BY clause or over event caches in SPLASH inside UDFs and FLEX operators.

For the SUM, COUNT, AVG, and valueInserted aggregation functions, the server can perform additive optimization, where the function executes in constant time. In such cases, the time it takes to perform an operation is the same regardless of group size.

In a SELECT statement, the server can perform additive optimization provided functions eligible for optimization are used in all values being selected, with the exception of the columns referenced in the GROUP BY clause.

The following SELECT is optimized for additive optimization since all non-GROUP BY columns (name, counter, summary) only use additive aggregation functions (that is, valueInserted, SUM, and COUNT).

```
CREATE OUTPUT WINDOW AggResult
    SCHEMA (id INTEGER, name STRING, counter INTEGER, summary FLOAT)
    PRIMARY KEY DEDUCED
AS
    SELECT BaseInput.intData_1 AS id,
        valueInserted(BaseInput.strData_1) AS name,
        count(BaseInput.intData_1) AS counter,
        sum(BaseInput.dblData_1) AS summary
FROM BaseInput
GROUP BY BaseInput.intData_1
;
```

Note: For optimal peformance, when selecting only the column in a SELECT statement with a GROUP BY clause, use the valueInserted function, where feasible.

The following SELECT is not optimized for additive optimization since one of the non-GROUP BY columns (name) directly selects a column which cannot be computed additively.

```
CREATE OUTPUT WINDOW AggResult
    SCHEMA (id INTEGER, name STRING, counter INTEGER, summary FLOAT)
    PRIMARY KEY DEDUCED
AS
    SELECT BaseInput.intData_1 AS id,
        BaseInput.strData_1 AS name,
        count(BaseInput.intData_1) AS counter,
        sum(BaseInput.dblData_1) AS summary
FROM BaseInput
GROUP BY BaseInput.intData_1
;
```

When applying aggregation functions over an event cache, additive optimization is turned on when using the SUM, COUNT, AVG, or valueInserted functions only in the ON clause of a FLEX operator. The additive optimization does not apply when functions are used inside a UDF.

The following Flex stream computes the SUM in the ON clause additively, since the SUM function is computed additively and the used EventCaches (e0,e1) are declared locally.

```
CREATE INPUT WINDOW In1
    SCHEMA (c1 INTEGER, c2 STRING, c3 INTEGER, summary FLOAT)
    PRIMARY KEY (c1, c2);

CREATE FLEX MyFlex
    IN In1
    OUT OUTPUT WINDOW FlexOut
    SCHEMA (c1 INTEGER, c2 INTEGER, c3 INTEGER, c4 INTEGER)
    PRIMARY KEY (c1, c2)
BEGIN
    declare
        eventCache(In1, coalesce) e0;
        eventCache(In1, coalesce) e1;
    end;

    ON In1 {
     {
        output setOpcode([c1=In1.c1;c2=In1.c2;|
c3=sum(e0.c1);c4=sum(e1.c3);],getOpcode(In1));
     }
    };
END;
```

The following Flex stream is not computed additively , since the STDDEV function cannot be computed additively.

```
CREATE INPUT WINDOW In1
    SCHEMA (c1 INTEGER, c2 STRING, c3 INTEGER)
    PRIMARY KEY (c1, c2);

CREATE FLEX MyFlex
    IN In1
    OUT OUTPUT WINDOW FlexOut
    SCHEMA (c1 INTEGER, c2 INTEGER, c3 INTEGER, c4 FLOAT)
    PRIMARY KEY (c1, c2)
BEGIN
    declare
        eventCache(In1, coalesce) e0;
        eventCache(In1, coalesce) e1;
    end;

    ON In1 {
     {
        output setOpcode([c1=In1.c1;c2=In1.c2;|
c3=sum(e0.c1);c4=stddev(e1.c3);],getOpcode(In1));
     }
    };
END;
```

Another restriction is that additive optimizations are disabled when functions are used inside nonlinear statements (if, while, for, and case statements). To enable additive optimizations when using a function within a nonlinear statement, assign the result of the function to a variable outside of the statement. Then use the variable inside the nonlinear statement.

Note: The function used within the nonlinear statement must be from the set of functions eligible for additive optimization.

The following SELECT is not optimized for additive optimization since one of the expressions (CASE) in the SELECT list is a nonlinear expression.

```
CREATE OUTPUT WINDOW AggResult
    SCHEMA (id INTEGER, name STRING, counter INTEGER, summary FLOAT)
    PRIMARY KEY DEDUCED
AS
    SELECT BaseInput.intData_1 AS id,
        valueInserted(BaseInput.strData_1) AS name,
       CASE WHEN (count(BaseInput.intDATA_1) < 100) THEN 0 ELSE 1 END
AS counter,
        sum(BaseInput.dblData_1) AS summary
FROM BaseInput
GROUP BY BaseInput.intData_1
;
```

Index

A

adapters 24
Adaptive Server database
    executing DDL 158
Adaptive Server Enterprise
    schema 155
aggregation
    GROUP BY clause 52
    GROUP FILTER clause 52
    GROUP ORDER BY clause 52
    HAVING clause 52
Aging Policy 8
ANSI syntax 49
ASE
    executing DDL 158
    generating DDL scripts 156
ASE database
    creating objects 158
    creating schema 155
ASE model 132
    creating with categories 139
    extending 134
    generating from ASE model 149
ASE physical data model
    generate from an ESP schema logical data model 147
ASE table
    generating schema 141
ASE.pdm 156
ASE.xem
    description 131
attribute properties
    length 142
    name 142
    precision 142
attributes 142
    ESP Data Type 135
    schema definition 143
auto checkpoint 96

B

basic project components
    queries 22
bigdatetime
    format codes 170
bindings 15
Boolean
    Boolean Literals 40
Boolean Literals
    Boolean Literals 40

C

calendar 174
calendar functions 174
case-insensitive 38
case-sensitive 38
categories
    changing 134, 135
    creating a model 139
    displaying 134
    ESP models 132, 134
    merging 134, 135
    PowerDesigner 132, 134
    setting up 134
category set
    ESP models 132
CCL
    advanced techniques overview 63
    creating a model 146
    language components 27
    order of elements 24
    overview 2
CCL keywords 159
CCR
    bindings examples 15
checking a model
    custom checks 144, 145
    data types 145
    indexes 145
    lookup information 145
    options 144
    PowerDesigner standard checks 145
    uniqueness 145
ckCount parameter 109
clauses
    GROUP BY 52
    GROUP FILTER 52
    GROUP ORDER BY 52
    HAVING 52
    MATCHING 51

UNION 45
WHERE 43
clustering
log stores 103
code names of tables 129
columns
BIGROWTIME 20
ROWID 20
ROWTIME 20
combining queries 45
comma-separated syntax 51
complex join example 51
configuration file 183
configuring
log store size 105
consistent recovery 95
continuous queries 7
count-based retention 8
CREATE SCHEMA statement 21

D

data definition language scripts
See DDL scripts
data flow 181
data model
PowerDesigner standard checks 145
validating 145
data models
integrated 127
modifying 155
overview 129
scenarios 127
tables 129
data partitioning
modules 68
streams 68
windows 68
data types
schema model checks 145
data-flow programming
example 1
introduction 1
database owner
changing from RAP_USER 155
databases
creating objects 155, 157, 158
generating schema 155
datatypes
supported datatypes in Event Stream Processor 27
date
format codes 170
date/time format codes 170
daylight saving time (DST) 161
DDL scripts
generating (overview) 155
executing for ASE 158
executing for IQ 157
executing for SAP HANA 157
executing generated script 158
for ASE 156
for HANA 156
for IQ 156
generating using Powerdesigner 155
generating using PowerDesigner 156
DDL, custom
SAP HANA 157
DDL, custom (Adaptive Server Enterprise)
executing 158
DDL, custom (Sybase IQ)
changing default database owner 155
executing 157
declaration
typedefs 64
variables 65
declare blocks
declaring 67
global 63
local 63
overview 63
declaring parameters
parameters 64
delete 119
delta streams 13, 14
bindings on 15
dependency loops 21
destination table name
schema definition attribute 142
DST 161

E

entity tool
creating schema definition 140
error streams 90
ESP 128, 129
overview 127
ESP Data Type
customizing 135
setting attribute manually 135

ESP model category
    enabling 135
ESP models
    category set
    See models
        See also ASE model
ESP Schema
    schema 155
ESP schema logical model
    generating a HANA model 147
    generating an ASE model 147
    generating an IQ model 147
ESP Schema model 129, 132
    creating with CCL 146
    generating from ASE model 149, 150
    generating from HANA model 149
    generating from IQ model 149, 150
    generating from physical data model 149, 150
    overview 129
ESP Schema model generating an ASE model
    generating an ASE model 149
ESP Schema model generating an IQ model
    generating an IQ model 149
ESP.mcc
    description 132
ESPSchema.xem
    description 131
event streams
    overview 5
events
    definition 5
    delete 6
    examples 5
    insert 6
    update 6
example projects 121
examples
    ANSI syntax 49
    comma-separated syntax 51
    complex join 49, 51
    complex stream-window join 49
    internal pulsing 113
    merging data 45
    order book 114
    pattern matching 51
    simple full outer join 49
    simple inner join 49
    simple left join 49
    simple stream-window left join 49
    using a UNION operator 45
executing SPLASH within CCL 117
expressions
    compound expressions 35
    simple expressions 35
extended definition files
    overview 131
extended model definitions
    See extensions
extending a model
    manually 134
extensions
    model 131
    overview 131
    transformations 131
    .xem files 131

F

files
    calendar 174
filtering data 43
flex operators 68
Flex operators 117
    accessing input streams 117
    accessing the event 117
    transaction blocks 119
    using output statements 119
foreign keys
    adding 149
    lineage analysis 154
format codes
    bigdatetime 170
    date 170
    date/time 170
    timestamp 170

G

generating a diagram
    impact analysis 152
    lineage analysis 152
guaranteed delivery 98
    and log stores 104
    CCR example - bindings 15
GUI authoring
    See visual authoring

H

HANA physical data model
    generate from an ESP schema logical data model 147
HANA.pdm 156
HANA.xem 131
hdbsql, HANA 157

I

identifiers
    defining 144
    overview 143
    primary keys 143, 144
impact analysis 133
    customizing 152
    example 153, 154
    foreign keys 154
    generating a diagram 152
    IAM 151
    launch 151
    overview 150
    preview 151
    printing 152
    refining 152
    reviewing 152
    saving to RTF or CSV 152
impact analysis model (IAM)
    generating 152
implicit
    columns 20
    windows 11
indexes
    editing 148
input 19
input streams
    accessing 117
insert 119
insert opcode
    and streams 7
Interactive SQL
    executing DDL scripts 157
internal pulsing 113
Interval
    Time Literals 39
INTERVAL
    Time Literals 39
Interval Literals
    Time Literals 39
intervals
    values 31
IQ database
    creating objects 157
    creating schema 155
IQ model 132
    creating with categories 139
    generating from IQ model 149
IQ physical datamodel
    generate from an ESP schema logical data model 147
IQ table
    generating schema 141
IQ.pdm 156
IQ.xem
    description 131

J

joins
    ANSI syntax 49
    cardinality 46
    complex join example 49, 51
    complex stream-window join example 49
    examples 46
    key field rules 48
    simple full outer join example 49
    simple inner join example 49
    simple left join example 49
    simple stream-window left join example 49
    types 46

K

KEEP clause
    retention policies 8
key field rules 48
keywords 159

L

length
    schema definition attribute 142
lineage analysis 133
    example 154
    foreign keys 154
    generating a diagram 152
    IAM 151
    launch 151

overview 150
preview 151
reviewing 152
Literals
Literals 39
Time Literals 39
log level 183
log store
features 89
log store loops 21
optimization techniques 90
state after recovery 89
log stores
and consistent recovery 95
and guaranteed delivery 104
creating 103
guidelines 104
managing 109
sizing 105, 109
logical data model
creating 139
extending 134
generating from physical data model 149, 150
logical data models
ESPSchema.xem 131
lookup columns
adding 149
schema definition attributes 142
lookup information
schema model checks 145

M

MATCHING clause 51
memory 183
memory store 21
merging
model categories 135
model
ESP Schema 129
sample 137
model categories
merging 135
modeling
scenarios 127
models 132
analyzing impact of changes 133, 150
analyzing object lineage 133, 150
creating with categories 139
creating with CCL 146
sample project 128
setting up ESP category set 134
modularity 86, 87
overview 85
module
create 86, 87
load 86, 87
parameters 87
use 86, 87
modules 68
rules for 85
monitoring 92

N

named schema 21
naming 38
Numeric
Numeric Literals 41

O

Object
Object Names 40
opcodes
defined 6
delete 6
insert 6
safedelete 6
update 6
upsert 6
operators
arithmetic operators 32
comparison operators 32
LIKE operators 32
logical operators 32
string operators 32
order book 114
output 19
Output parameter on input streams, windows 15
output statements
using with Flex operators 119
overview 2

P

parameters 64
for sizing log stores 109
in modules 87

initializing parameters at runtime 64
PARAMETERS clause
parameter binding 87
partitioning
in ASE model 148
joins 83
retention policies 78
windows 78
partitioning data
modules 68
streams 68
windows 68
pattern matching 51
performance
count-based retention 8
improving 177
SLACK value 8
persistence
log store 21, 89
physical data model
generating a logical data model 149, 150
physical options
in table properties 148
PowerDesigner
creating Adaptive Server database schema 155
creating IQ database schema 155
creating IQ schema 155
documentation 128
modeling overview 129
producing DDL 156
sample project 128
tutorials 128
PowerDesigner categories
See categories
PowerDesigner extensions
setting up 134
precision
schema definition attribute 142
primary keys
identifiers in schema 144
multicolumn 143
projects
configuring data flow between 15
development task flow 25
example 121
properties
schema definition, general tab 142
properties sheet
defining schema 141

Q

queries
basic syntax 22
GROUP BY clause 52
GROUP FILTER clause 52
GROUP ORDER BY clause 52
HAVING clause 52
MATCHING clause 51
UNION operator 45
WHERE clause 43
Queries
reference table 54
Query
reference table 55
query construction
aggregating data 43
combining queries 43
filtering data 43
joining multiple datasources 43
using pattern-matching rules 43

R

reconnection interval for bindings
setting in CCR 15
Reference table queries 54
Reference table query 55
Reference Table Query
in SPLASH 58, 60
reservePct parameter 109
retention
count-based 8
semantics 8
time-based 8
retention policies
description 8
retention semantics 8

S

safedelete
defined 6
sample model
exploring 137
opening 128, 137

overview 137
sample project
opening 128
PowerDesigner 128
SAP HANA
generating DDL scripts 156
SAP HANA database
creating objects 157
SAP Sybase IQ
generating DDL scripts 156
schema 21
Adaptive Server database 155
creating a schema definition 140, 141
creating for Adaptive Server Enterprise database 155
creating for IQ database 155
creating with PowerDesigner 155
generating from a table 141
IQ database 155
SCHEMA clause 21
schema definition
adding 140
adding attributes 143
attributes 132
creating 140, 141
creating with entity tool 140
identifiers 132
overview 132
properties 132
schema definition properties
attributes tab 142
destination table name 142
general tab 142
name 142
schema type 142
schema definitions
definining properties 141
editing 141
list 133
properties sheet 141
schema model
creating with categories 139
custom checks 144, 145
data types 145
lookup information 145
standard checks 145
uniqueness 145
validating 144, 145
Schema model
See ESP Schema model
scope
for modules 85
sizing
log stores 109
sizing log stores 105
SLACK
count-based retention 8
performance 8
SPLASH
overview 3
Reference Table Query in 58, 60
SPLASH examples
internal pulsing 113
order book 114
SPLASH routines 113
SSL
configuring bindings for 15
stateful elements 7
statements
CREATE LOG STORE 21
CREATE MEMORY STORE 21
CREATE MODULE 86, 87
IMPORT 86, 87
LOAD MODULE 86, 87
stores
log store 21, 89
log stores 103
memory store 21
stream 183
streams 7, 14, 68, 92
bindings on 15
error 90
input 19
input, configuring to provide output 15
local 19
output 19
schema 21
structure 21
String
String Literals 40
String Literals
String Literals 40
Studio
overview 3
sweepamount parameter 109
Sybase IQ
schema 155

Sybase IQ model
- extending 134

T

table
- finding in model 129

tables
- Bond History 129
- Bond Quote 130
- Bond Trade 130
- code names 129
- descriptions 129
- Dividend Event 130
- Index History 130
- Index Intraday 130
- Mutual Fund History 130
- Option History 130
- Option Quote 130
- Option Trade 130
- Split Event 131
- Stock History 131
- Stock Quote 131
- Stock Trade 131

text authoring
- overview 3

throughput 181
- maximizing 177

Time
- Time Literals 39

Time Literals
- Time Literals 39

time zones 161, 162

time-based retention 8

timestamp
- format codes 170

Timestamp
- Time Literals 39

transaction blocks 119

transformation rules
- in model generation 146

transformations
- defined in extension files 131

typedefs 64

U

UNION operator 45

unions 45

uniqueness
- schema model checks 145

unnamed windows 11

update 119

upsert
- defined 6

usage 183

V

validating
- schema model 144

variables 65

visual authoring
- overview 3

W

WHERE clause 43

windows 14, 68
- bindings on 15
- implicit 11
- input 11, 19
- join 11
- local 11, 19
- named 7, 11
- output 11, 19
- persistence 89
- schema 21
- structure 21
- unnamed 7, 11

Writing SPLASH 113

X

.xem files
- overview 131

Z

zero data loss 95
- achieving 100
- auto checkpoint 96
- consistent recovery 95
- guaranteed delivery 98