



**Examples Guide**

---

**SAP Sybase Event Stream  
Processor 5.1 SP03**

DOCUMENT ID: DC01683-01-0513-01

LAST REVISED: August 2013

Copyright © 2013 by SAP AG or an SAP affiliate company. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice.

Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors. National product specifications may vary.

These materials are provided by SAP AG and its affiliated companies ("SAP Group") for informational purposes only, without representation or warranty of any kind, and SAP Group shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP Group products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and other countries. Please see <http://www.sap.com/corporate-en/legal/copyright/index.epx#trademark> for additional trademark information and notices.

# Contents

<b>CHAPTER 1: Learning CCL by Example .....</b>	<b>1</b>
<b>CHAPTER 2: Adapters Examples .....</b>	<b>3</b>
ATTACH ADAPTER Statement .....	3
ADAPTER START GROUPS Statement .....	4
Schema Inheritance .....	4
Adapter Data with Opcodes .....	5
File CSV Output Adapter .....	6
Database Input Adapter .....	7
Database Output Adapter .....	8
Database Input Adapter with Polling .....	9
SAP HANA Output Adapter .....	11
<b>CHAPTER 3: Stream and Window Examples .....</b>	<b>13</b>
Streams .....	13
Local Windows and Output Windows .....	14
Delta Stream .....	14
Join Windows .....	15
Join Streams .....	16
Outer Join .....	17
Union Streams .....	18
Jumping Windows .....	19
Splitter .....	20
<b>CHAPTER 4: Function Examples .....</b>	<b>23</b>
CREATE LIBRARY statement .....	23
Aggregate Functions .....	24
Bitwise Functions .....	24

Data Aggregation .....	25
<b>CHAPTER 5: Store Examples .....</b>	<b>27</b>
STORES .....	27
<b>CHAPTER 6: Flex Examples .....</b>	<b>29</b>
Data Management with Flex Streams .....	29
Multiple Inputs .....	30
Average Trade Price with Timer .....	31
Variables in the DECLARE Block .....	32
Event Cache .....	33
SPLASH with if/then/else .....	34
SPLASH with getOpcode .....	35
<b>CHAPTER 7: DECLARE Block Examples .....</b>	<b>37</b>
CCL Function .....	37
Parameter Declaration .....	38
<b>CHAPTER 8: Data Selection Examples .....</b>	<b>39</b>
AGING Column .....	39
AGING Column with Time Option .....	40
Data Aggregation .....	40
Data Aggregation with Filter .....	41
GROUP BY Clause with last() Function .....	41
KEEP Clause .....	43
KEEP Clause with AGING Clause .....	43
KEEP ALL Clause .....	44
KEEP LAST clause .....	44
KEEP PER Clause .....	45
KEEP UNTIL Clause .....	46
Filter with WHERE Clause .....	47
MATCHING clause .....	47

<b>Matching a Sequence of Events .....</b>	<b>48</b>
<b>Matching Non-Events .....</b>	<b>49</b>
<b>Row Time .....</b>	<b>49</b>
<b>AUTOGENERATE Clause .....</b>	<b>50</b>
<b>CHAPTER 9: Module Examples .....</b>	<b>53</b>
<b>CREATE MODULE .....</b>	<b>53</b>
<b>Load Module .....</b>	<b>54</b>
<b>CHAPTER 10: Advanced Examples .....</b>	<b>55</b>
<b>Portfolio Valuation .....</b>	<b>55</b>
<b>Trades Log .....</b>	<b>56</b>
<b>Vectors and Dictionaries .....</b>	<b>59</b>
<b>Dictionary of Dictionaries .....</b>	<b>60</b>
<b>Index .....</b>	<b>63</b>

# Contents

# CHAPTER 1      **Learning CCL by Example**

This guide is intended as a companion reference to the CCL examples included with SAP® Sybase® Event Stream Processor.

This guide describes the sequence of CCL elements used to achieve specific tasks within projects, using sample code to highlight the most relevant pieces of code to the task. By default, example files and the data files they read from are in C:\<installation directory>\ESP-5\_1\studio\learning directory. You can configure this directory during installation.

There are examples of simple projects available in SAP Sybase Event Stream Processor Studio that are not described in this guide. You can load and run them from the Samples link on the Welcome page.





Event Stream Processor includes several adapter-related CCL examples that demonstrate a range of functionality, including how to attach an adapter and perform schema inheritance.

---

**Note:** The example syntax occasionally wraps due to space constraints. Wrapped lines should be entered on a single line.

---

## ATTACH ADAPTER Statement

---

Use the **ATTACH ADAPTER** statement to attach a File CSV Input adapter to a window.

The example creates a schema named `TradeSchema` and an input window named `TradeWindow` that references the schema.

The example then attaches the File CSV Input adapter to `TradeWindow`.

This **ATTACH ADAPTER** instance is named `csvInConn1`, but you can assign it any name. The `TYPE` requirement refers to the adapter ID, which is unique to the adapter. The ID for the File CSV Input adapter is `dsv_in`. The example defines values for adapter parameters, either maintaining the default values or modifying them as needed. You can find the adapter type or ID and a list of parameters for each adapter in the *Adapters Guide*.

```
ATTACH INPUT ADAPTER csvInConn1
TYPE dsv_in
TO TradeWindow
PROPERTIES
  blockSize=1,
  dateFormat='%Y/%m/%d %H:%M:%S',
  delimiter=',',
  dir='$ProjectFolder/./data',
  expectStreamNameOpcode=false,
  fieldCount=0,
  file='stock-trades.csv',
  filePattern='*.csv',
  hasHeader=true,
  safeOps=false,
  skipDels=false,
  timestampFormat= '%Y/%m/%d %H:%M:%S';
```

## ADAPTER START GROUPS Statement

---

Use the **ADAPTER START GROUPS** statement to specify a start order for adapters in a project.

The example creates schemas named `TradeSchema`, `CompanySchema`, and `JoinSchema` inherits its schema from `TradeSchema`. The text in parentheses tells the project server to extend `TradeSchema` by adding another column named `Company`.

```
Create Schema JoinSchema
    inherits TradeSchema (Company String);
```

The example creates an input window named `TradeWindow` that references `TradeSchema`, and another input window named `CompanyInfo` that references `CompanySchema`. An output join window that uses the structure defined in `JoinSchema` is created to join the `TradeWindow` and `CompanyInfo` input windows using their symbol and timestamp values.

```
CREATE OUTPUT WINDOW Join1
    SCHEMA JoinSchema Primary Key deduced
    AS
    SELECT t.Ts as Ts, c.StockSymbol as Symbol ,
    t.Price as Price , t.Volume as Volume, c.Company as Company
    FROM TradeWindow t join CompanyInfo c
    on t.Symbol = c.StockSymbol
    group by t.Ts
    ;
```

The example attaches a File CSV Input adapter named `csvTradesIn2` to `TradeWindow`, and another File CSV Input adapter named `csvCompanyIn` to `CompanyInfo`. The adapter instance named `csvTradesIn2` is assigned to `RunGroup0`, and the adapter instance named `csvCompanyIn` is assigned to `RunGroup1`.

The **ADAPTER START GROUPS** statement uses these adapter group assignments when specifying the order in which adapters start. In this example, the project server starts `RunGroup1` adapters first, followed by `RunGroup0` adapters.

```
ADAPTER START GROUPS RunGroup1, RunGroup0 ;
```

## Schema Inheritance

---

Tell a new schema to inherit the structure of an existing schema.

The example creates a schema named `TradeSchema`.

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

The example then creates the schema `VTradeSchema`, and uses the **INHERITS** syntax to extend `VTradeSchema` by incorporating `TradeSchema` column values.

```
CREATE SCHEMA VTradeSchema INHERITS TradeSchema (vwap money(4));
```

The example creates an input window named `TradeWindow`, to which it attaches the File CSV Input adapter.

Finally, the example creates an aggregate output window named `VwapWindow`, in which the volume-weighted average price is returned for `TradeWindow` data. The return values are grouped by `Symbol`.

```
CREATE OUTPUT WINDOW VwapWindow
  SCHEMA VTradeSchema
  PRIMARY KEY DEDUCED
AS
  SELECT TradeWindow.Ts Ts,
         TradeWindow.Symbol AS Symbol,
         TradeWindow.Price Price,
         TradeWindow.Volume Volume,
         ((SUM(TradeWindow.Price*TradeWindow.Volume)) /
          (SUM(TradeWindow.Volume))) AS vwap
  FROM TradeWindow
  GROUP BY TradeWindow.Symbol;
```

## Adapter Data with Opcodes

---

Use the **expectStreamNameOpcode** adapter property for the File CSV Input adapter.

The example uses the following data set:

```
win1,i,1,abc, row1
win1,i,2,zzzz, row2
win1,i,3,dfp, row3
win1,d,1,abc, row1
win1,u,3,dfp12, row3a
```

The `i`, `d`, and `u` values in the data are opcodes for inserting, deleting, and updating data, respectively.

The example creates an input window for the data named `win1`, to which it attaches the File CSV Input adapter.

The adapter property `expectStreamNameOpcode` is set to `true` so that the project server knows there are opcodes in the incoming data that it must execute.

```
Input Window
CREATE INPUT WINDOW win1
  SCHEMA (
    a integer,
```

```

    b string ,
    c string )
PRIMARY KEY (a);

```

```

Input Adapter
ATTACH INPUT ADAPTER csvInConn1
  TYPE dsv_in
  TO win1
  PROPERTIES expectStreamNameOpcode = TRUE ,
  dir='../exampledata',
  file = 'input1.csv' ;

```

## File CSV Output Adapter

Use the File CSV Output adapter to send data to an external destination.

The example creates a schema named TradeSchema that is referenced by an input window named InTrades. The example attaches a File CSV Output adapter named csvOut and a File CSV Input adapter named InConn to InTrades.

```

ATTACH OUTPUT ADAPTER csvOut
  TYPE dsv_out
  TO InTrades
  PROPERTIES prependStreamNameOpcode = FALSE ,
  dir = '../exampleoutput' , file = 'csvoutput.csv' ,
  outputBase = FALSE , delimiter = ',' , hasHeader = FALSE ,
filePattern = '*.csv' ,
  onlyBase = FALSE , dateFormat = '%Y-%m-%dT%H:%M:%S' ,
  timestampFormat = '%Y-%m-%dT%H:%M:%S' ;

ATTACH INPUT ADAPTER InConn
  TYPE dsv_in
  TO InTrades
  PROPERTIES expectStreamNameOpcode = FALSE ,
  fieldCount = 0 ,
  dir = '../exampledata' ,
  file = 'stock-trades.csv' ,
  repeatCount = 0 , repeatField = '-' ,
  delimiter = ',' , hasHeader = FALSE ,
  filePattern = '*.csv' , pollperiod = 0 ,
  safeOps = FALSE , skipDels = FALSE , dateFormat = '%Y/%m/%d
%H:%M:%S' ,
  timestampFormat = '%Y/%m/%d %H:%M:%S' ,
  blockSize = 1 ;

```

## Database Input Adapter

---

Use the Database Input adapter to connect to a database.

### *Prerequisites*

To run this example, create a `Trades` table in your database using the supported syntax. The table should include these values:

Column	Datatype	Value
Ts	datetime	not null
Symbol	char(4)	not null
Price	money	not null
Volume	int	not null

You must also create a unique index named `ind1` on `Trades (Ts)` and grant all permissions on `Trades` to `public`.

Finally, configure the `services.xml` file in `<ESP_HOME>/bin` using this example as a model:

```
<Service Name="dbExample" Type="DB">
    <Parameter
Name="DriverLibrary">esp_db_jdbc_sybase_lib</Parameter>
    <Parameter Name="Host">mydbserver</Parameter>
    <Parameter Name="Port">5000</Parameter>
    <Parameter Name="User">test4</Parameter>
    <Parameter Name="Password">password</Parameter>
    <Parameter Name="Database">interpubs</Parameter>
    <Parameter Name="ConnectionString"></Parameter>
    <Parameter Name="ConnectionPoolSize">-1</Parameter>
</Service>
```

Populate the table with data before running the example.

### *Example*

The example creates a schema named `TradeSchema`, followed by an input window named `TradeWindow` and an output window named `TradeOutWindow` that each reference `TradeSchema`. **SELECT** all (\*) syntax tells the project server to output all data processed by `TradeWindow` to `TradeOutWindow`.

The example attaches a Database Input adapter to `TradeWindow` to read data from the database you set up as a prerequisite.

```
ATTACH INPUT ADAPTER dbInConn1
TYPE db_in
TO TradeWindow
PROPERTIES service = 'dbExample' ,
  query = 'Select * from Trades' ,
  table = 'Trades' ,
  pollperiod =0 ,
  dateFormat = '%Y-%m-%d %H:%M:%S' , timestampFormat = '%Y-%m-%d %H:
%M:%S' ;
```

## Database Output Adapter

Use a Database Output adapter to send data to an external database.

### Prerequisites

To run this example, create a `VwapWindow` table in your database using the supported syntax.

The table should include these values:

Column	Datatype	Value
Symbol	char(4)	not null
Price	money	not null

You must also create a unique index named `ind1` on `Trades (Ts)` and grant all permissions on `VwapWindow` to `public`.

Finally, configure the `services.xml` file in `<ESP_HOME>/bin` using the following example as a model for configuration:

```
<Service Name="dbExample" Type="DB">
  <Parameter
Name="DriverLibrary">esp_db_jdbc_sybase_lib</Parameter>
  <Parameter Name="Host">mydbserver</Parameter>
  <Parameter Name="Port">5000</Parameter>
  <Parameter Name="User">test4</Parameter>
  <Parameter Name="Password">password</Parameter>
  <Parameter Name="Database">interpubs</Parameter>
  <Parameter Name="ConnectionString"></Parameter>
  <Parameter Name="ConnectionPoolSize">-1</Parameter>
</Service>
```

The table is automatically populated with data from the File CSV Input adapter.

### Example

The example creates a schema named `TradeSchema`, followed by an input window named `TradeWindow` that references `TradeSchema`.

The example creates an aggregate output window named `VwapWindow`, in which the volume weighted average price is returned for `TradeWindow` data. The return values are grouped by `Symbol`.

```
CREATE    output WINDOW VwapWindow
SCHEMA (Symbol STRING, vwap MONEY(2))
  PRIMARY KEY DEDUCED
  AS
SELECT TradeWindow.Symbol AS Symbol,
((SUM(TradeWindow.Price * TradeWindow.Volume)) /
(SUM(TradeWindow.Volume))) AS vwap
FROM TradeWindow
GROUP BY TradeWindow.Symbol;
```

The example attaches a Database Output adapter to `VwapWindow`. The project server processes date values in date format, which means date values are truncated.

```
ATTACH OUTPUT ADAPTER dbOutConn1 TYPE db_out TO VwapWindow
PROPERTIES service = 'dbExample' ,
  table = 'VwapWindow' , outputBase = FALSE , truncateTable = TRUE ,
dateFormat = '%Y-%m-%d %H:%M:%S' , timestampFormat = '%Y-%m-%d %H:%M:
%S' , onlyBase = FALSE , batchLimit = 1 ;
```

The example attaches a File CSV Input adapter to `TradeWindow` to read data from an external source and populate the database you set up as a prerequisite.

```
ATTACH INPUT ADAPTER csvInConn1
TYPE dsv_in
TO TradeWindow
PROPERTIES
blockSize=1,
dateFormat='%Y/%m/%d %H:%M:%S',
delimiter=',',
dir='../exampledata',
expectStreamNameOpcode=false,
fieldCount=0,
file='stock-trades.csv',
filePattern='*.csv',
hasHeader=true,
safeOps=false,
skipDels=false,
timestampFormat= '%Y/%m/%d %H:%M:%S';
```

## Database Input Adapter with Polling

---

Use a Database Input adapter to connect to and poll a database.

### *Prerequisites*

To run this example, create a `Trades` table in your database using the supported syntax. The table should include these values:

## CHAPTER 2: Adapters Examples

Column	Datatype	Value
Ts	datetime	not null
Symbol	char(4)	not null
Price	money	not null
Volume	int	not null

You also need to create a unique non-clustered index called `ind1` on `Trades (Ts)`, and grant all permissions on `Trades` to `public`.

Finally, configure the `services.xml` file in `<ESP_HOME>/bin` using the following example as a model for configuration:

```
<Service Name="dbExample" Type="DB">
    <Parameter
Name="DriverLibrary">esp_db_jdbc_sybase_lib</Parameter>
    <Parameter Name="Host">mydbserver</Parameter>
    <Parameter Name="Port">5000</Parameter>
    <Parameter Name="User">test4</Parameter>
    <Parameter Name="Password">password</Parameter>
    <Parameter Name="Database">interpubs</Parameter>
    <Parameter Name="ConnectionString"></Parameter>
    <Parameter Name="ConnectionPoolSize">-1</Parameter>
</Service>
```

Populate the table with data, then run the example.

### *Example*

The example creates a schema named `TradeSchema`, followed by an input window named `TradeWindow` and output window named `TradeOutWindow` that each reference `TradeSchema`. **SELECT** all (\*) syntax outputs all data processed by `TradeWindow` to `TradeOutWindow`.

The example attaches a Database Input adapter to `TradeWindow` to read data from the database you set up as a prerequisite. A poll period of 10 for this adapter instance means that the database is polled for new content every 10 seconds.

```
ATTACH INPUT ADAPTER dbInConn1
TYPE db_in
TO TradeWindow
PROPERTIES service = 'dbExample' ,
  query = 'Select * from Trades' ,
  table = 'Trades' ,
  pollperiod =0 ,
```



```
dateFormat = '%Y-%m-%d %H:%M:%S' , timestampFormat = '%Y-%m-%d %H:
%M:%S' ;
```

## SAP HANA Output Adapter

Use the SAP HANA Output adapter to send data from Event Stream Processor to an SAP HANA database.

### Prerequisites

To run this example, create a table named HANARANDOM in the SAP HANA database. The table should include these columns:

Column	Datatype
RANDOMINT	INTEGER

Once you have created the table, define an Open Database Connectivity (ODBC) data source that points to your SAP HANA server.

Finally, configure the `services.xml` file in `<ESP_HOME>/bin` using the following example as a model for configuration. The `DriverLibrary` parameter for this example (`esp_db_odbc_lib`) works only for Windows. Use `esp_db_odbc64_lib` for all other platforms. Replace the `hanadatasource` from this example with the data source you specified when you created the ODBC datasource. Also replace the `HANA_USER` and `password` from this example with your own username and password:

```
<Service Name="hanaservice" Type="DB">
  <Parameter Name="DriverLibrary">esp_db_odbc_lib</Parameter>
  <Parameter Name="DNS">hanadatasource</Parameter>
  <Parameter Name="User">HANA_USER</Parameter>
  <Parameter Name="Password"encrypted="false">password</
Parameter>
</Service>
```

The table is automatically populated with data from an input adapter named `RandomIn`.

See the *Adapters Guide* for more information about how to set up the SAP HANA Output adapter.

### Example

The example creates a schema named `HANARandom`, followed by an input stream named `Newstream` that references `HANARandom`. `Newstream` attaches to a `RandomIn` input adapter. The `RandomIn` Input adapter provides random data as input to `Newstream`. `Newstream` also attaches to an SAP HANA Output adapter which uses the data provided by `RandomIn` to populate the database you set up as a prerequisite. The CCL code for this example is:

```
CREATE SCHEMA HANARANDOM (RANDOMINT integer);
CREATE INPUT STREAM NEWSTREAM SCHEMA HANARANDOM;
ATTACH INPUT ADAPTER RandomIn TYPE randomtuplegen_in TO NEWSTREAM
PROPERTIES RowCount=10;
```

## CHAPTER 2: Adapters Examples

```
ATTACH OUTPUT ADAPTER HANAOut TYPE hana_out TO NEWSTREAM PROPERTIES  
service='hanaservice', table='HANARANDOM';
```

Event Stream Processor includes several stream and window examples that demonstrate a range of functionality, including how to use delta streams, make joins and unions, and split streams.

---

**Note:** The example syntax occasionally wraps due to space constraints. Wrapped lines should be entered on a single line.

---

## Streams

---

Create input and local streams.

The example creates an input stream named `TradeStream` and a local stream named `TradeLocalStream`. The local stream uses **SELECT** all (\*) syntax to retrieve all data columns from `TradeStream`.

```
CREATE LOCAL STREAM TradeLocalStream
  SCHEMA (Ts BIGDATETIME, Symbol STRING, Price MONEY(2), Volume
  INTEGER)
```

```
AS
SELECT * from TradeStream;
```

```
ATTACH INPUT ADAPTER csvInConn1
TYPE dsv_in
TO TradeStream
PROPERTIES
  blockSize=1,
  dateFormat='%Y/%m/%d %H:%M:%S',
  delimiter=',',
  dir='../exampledata',
  expectStreamNameOpcode=false,
  fieldCount=0,
  file='stock-trades.csv',
  filePattern='*.csv',
  hasHeader=true,
  safeOps=false,
  skipDels=false,
  timestampFormat= '%Y/%m/%d %H:%M:%S';
```

The example attaches the File CSV Input adapter to `TradeStream`, then creates an output stream named `TradeOutputStream`.

```
CREATE OUTPUT STREAM TradeOutputStream
AS
  SELECT * FROM TradeLocalStream ;
```

## CHAPTER 3: Stream and Window Examples

TradeOutputStream retrieves all the data columns from TradeLocalStream using **SELECT** all syntax, and outputs them using the File CSV Output adapter.

```
ATTACH OUTPUT ADAPTER Adapter1
  TYPE dsv_out
  TO TradeOutputStream
  PROPERTIES
    dir = '../output' , file = 'streams.csv' , outputBase = TRUE ,
    hasHeader = TRUE ;
```

### Local Windows and Output Windows

---

Compare streams with windows and observe the differences between local and output windows.

The example creates a schema named TradeSchema, then an input window named TradeWindow that references TradeSchema. The File CSV Input adapter is attached to TradeWindow.

The example then creates a series of local and output streams and windows. The output stream and window are public; they communicate with external data sources using adapters. Local streams and windows are viewed only internally and cannot have adapters attached to them.

```
CREATE LOCAL STREAM LocalStream
  AS SELECT * FROM TradeWindow ;

CREATE OUTPUT STREAM OutputStream
  AS SELECT * FROM TradeWindow ;

CREATE LOCAL WINDOW LocalWindow
  PRIMARY KEY DEDUCED
  AS SELECT * FROM TradeWindow ;

CREATE OUTPUT WINDOW OutputWindow
  PRIMARY KEY DEDUCED
  AS SELECT * FROM TradeWindow ;
```

### Delta Stream

---

A delta stream incorporates the **getrowid** and **now** functions.

The example creates an input window named TradesWindow, to which it attaches the File CSV Input adapter.

The example then creates a delta stream named DeltaTrades and uses the **SELECT** clause to apply the **getrowid** and **now** functions to TradesWindow.

The **getrowid** function retrieves the sequence number of the rows for share symbol, timestamp, price, and value in the input window. The **now** function publishes the process date in `bigdatetime` format.

```
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
```

The example creates an output window named `TradesOut` for viewing the results.

## Join Windows

---

Use the **FROM** clause with ANSI **JOIN** syntax to join two windows.

The example creates two schemas named `StocksSchema` and `OptionsSchema`, and an output schema named `OutSchema`.

The example then creates two input windows named `InStocks` and `InOptions`, which use the structures defined in `StocksSchema` and `OptionsSchema`, respectively.

Finally, the example creates an output join window that uses the structure defined in `OutSchema` to join the `InStocks` and `InOptions` input windows using their symbol and timestamp values.

```
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 ;
```

### Join Streams

---

Join two windows into a stream.

The example creates two schemas named `StocksSchema` and `OptionsSchema`, followed by an input window named `InStocks` that references `StocksSchema`, and an input window named `InOptions` that references `OptionsSchema`.

The example creates an output join stream named `OutStockOption` that joins the `InStocks` and `InOptions` input windows using their symbol values.

```
CREATE OUTPUT STREAM OutStockOption AS
    SELECT InStocks.Ts Ts ,
           InStocks.Symbol Symbol ,
           InStocks.Price StockPrice ,
           InStocks.Volume StockVolume ,
           InOptions.StockSymbol OptionStockSymbol ,
InOptions.OptionSymbol OptionSymbol ,
           InOptions.Price OptionPrice,
           InOptions.Volume OptionVolume
    FROM InStocks JOIN InOptions
    on      InStocks.Symbol = InOptions.StockSymbol
;
```

The example creates two **ATTACH ADAPTER** instances named `csvInConn1` and `csvInOptions`. A File CSV Input adapter is attached to the `InStocks` window in one instance, and the `InOptions` window in another instance.

Finally, the example attaches a File CSV Output adapter named `Adapter1` to `OutStockOptions` to publish the results of the join stream.

```
ATTACH OUTPUT ADAPTER Adapter1
TYPE dsv_out
TO OutStockOption
PROPERTIES
    dir='../exampleoutput',
    file = 'joinstream.csv' ,
    outputBase =TRUE ,
    hasHeader = TRUE
;
```

## Outer Join

---

Create left, right, and full joins between input windows.

The example creates two schemas named `StocksSchema` and `OptionsSchema`. It then creates an input window named `InStocks` that references `StocksSchema`, and another input window named `InOptions` that references `OptionsSchema`.

The example creates an output window named `OutStockOptionFOJ` that creates a full join between `InStocks` and `InOptions` using their timestamp values.

```
CREATE OUTPUT WINDOW OutStockOptionFOJ
  PRIMARY KEY (Ts)
AS
  SELECT InStocks.Ts Ts , InStocks.Symbol Symbol , InStocks.Price
  StockPrice ,
         InStocks.Volume StockVolume , InOptions.StockSymbol
  OptionStockSymbol ,
         InOptions.OptionSymbol OptionSymbol , InOptions.Price
  OptionPrice,
         InOptions.Volume OptionVolume
  FROM InStocks FULL JOIN InOptions
  ON
  InStocks.Ts = InOptions.Ts;
```

The example creates an output window named `OutStockOptionLOJ` that creates a left outer join between `InStocks` and `InOptions` using their timestamp values.

```
CREATE OUTPUT WINDOW OutStockOptionLOJ
  Primary Key (Ts)
AS
SELECT InStocks.Ts Ts , InStocks.Symbol Symbol ,
       InStocks.Price StockPrice , InStocks.Volume StockVolume ,
       InOptions.StockSymbol OptionStockSymbol ,
       InOptions.OptionSymbol OptionSymbol , InOptions.Price
  OptionPrice,
       InOptions.Volume OptionVolume
FROM InStocks JOIN InOptions
  ON
  InStocks.Ts = InOptions.Ts ;
  Primary Key (Ts)
AS
SELECT InStocks.Ts Ts , InStocks.Symbol Symbol ,
       InStocks.Price StockPrice , InStocks.Volume StockVolume ,
       InOptions.StockSymbol OptionStockSymbol ,
       InOptions.OptionSymbol OptionSymbol , InOptions.Price
  OptionPrice,
       InOptions.Volume OptionVolume
FROM InStocks JOIN InOptions
  on
  InStocks.Ts = InOptions.Ts ;
```

## CHAPTER 3: Stream and Window Examples

The example creates an output window named `OutStockOptionROJ` that creates a right outer join between `InStocks` and `InOptions` using their timestamp values.

```
CREATE OUTPUT WINDOW OutStockOptionROJ
  PRIMARY KEY (Ts)
AS
SELECT InOptions.Ts Ts , InStocks.Symbol Symbol ,
       InStocks.Price StockPrice , InStocks.Volume StockVolume ,
       InOptions.StockSymbol OptionStockSymbol ,
       InOptions.OptionSymbol OptionSymbol , InOptions.Price
OptionPrice,
       InOptions.Volume OptionVolume
FROM InStocks RIGHT JOIN InOptions
  on
  InStocks.Ts = InOptions.Ts ;
```

The example attaches a File CSV Input adapter named `csvInStocks` to `InStocks`, and a File CSV Input adapter named `csvInOptions` to `InOptions`.

## Union Streams

---

Create a simple union between two windows.

The example creates two schemas named `StocksSchema` and `OptionsSchema` which define the structure for two input windows named `InStocks` and `InOptions`, respectively.

The example then creates an output window named `Union1` that creates a union between the `InStocks` and `InOptions` input windows.

```
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
;
```

The example concludes by creating two **ATTACH ADAPTER** instances named `csvInConn1` and `csvInConn2`. A File CSV Input adapter is attached to the `InStocks` window in one instance, and the `InOptions` window in another instance.



## Jumping Windows

---

Jumping Windows retain data for a specified interval of time or for a specified number of rows and delete all retained rows when the specified interval of time expires, or the specified number of rows is exceeded.

Jumping Windows are specified by the `KEEP EVERY` clause. A retention policy can be directly specified on a Window and indirectly specified on Windows and Delta Streams (using unnamed windows).

---

**Note:** Retention cannot be specified directly or indirectly on a Stream.

---

Tuples are the sets of data that are retained in the Window. Insert tuples affect retention, yet the arrival of update and/or delete tuples does not trigger the retention mechanism.

### Example

The example creates a schema named `TradesSchema` and applies that schema to the input window `Trades`.

```
CREATE SCHEMA TradesSchema (
  Id      integer,
  Symbol  string,
  Price   float,
  Volume  integer
) ;
```

```
CREATE INPUT WINDOW Trades
  SCHEMA TradesSchema
  PRIMARY KEY (Id) ;
```

The example then creates various types of Jumping Windows.

This creates a Jumping Window named `Every5Rows` from the source stream `Trades`. This window retains a maximum of five rows then deletes all five retained rows on the arrival of a new row.

```
CREATE OUTPUT WINDOW Every5Rows
  PRIMARY KEY DEDUCED
  KEEP EVERY 5 ROWS
  AS SELECT * FROM Trades ;
```

This creates a Jumping Window named `Every5Seconds` from the source stream `Trades`. This window retains rows for a maximum of five seconds then deletes all retained rows when the time interval expires.

```
CREATE OUTPUT WINDOW Every5Seconds
  PRIMARY KEY DEDUCED
  KEEP EVERY 5 SECONDS
  AS SELECT * FROM Trades ;
```

## CHAPTER 3: Stream and Window Examples

This creates an unnamed Jumping Window from the source stream `Trades`. This window retains a maximum of five rows for each unique value of `Symbol` then deletes all five retained rows upon the arrival of a sixth row with the same `Symbol` value.

```
CREATE OUTPUT WINDOW Every5RowsPerSymbol
PRIMARY KEY DEDUCED
AS SELECT * FROM Trades KEEP EVERY 5 ROWS PER(Symbol)
```

The example concludes by attaching the XML Input Adapter to `Trades` to process the incoming stream data.

```
ATTACH INPUT ADAPTER xmlInConn1
TYPE xml_in
TO Trades
PROPERTIES
    blockSize=1,
    dir='../exampledata',
    file='Trades.xml',
    filePattern='*.xml',
    safeOps=false,
    skipDels=false ;
```

## Splitter

---

Use the splitter feature to route data from one stream to multiple streams.

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. `IBM_MSFT_Tradeswin` retains data with the symbols `IBM` or `MSFT`. `Large_TradesWin` retains all data where the product of `trw.Price * trw.Volume` is greater than 25,000. `Other_Trades` retains all data sets that do not meet the conditions placed on the two previous output windows.

```
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 ;
```

The example concludes by attaching the XML Input Adapter to Trades to process the incoming stream data.

```
ATTACH INPUT ADAPTER xmlInConn1
  TYPE xml_in
  TO Trades
  PROPERTIES
    blockSize=1,
    dir='../exampledata',
    file='Trades.xml',
    filePattern='*.xml',
    safeOps=false,
    skipDels=false
```



## CHAPTER 4      **Function Examples**

Event Stream Processor includes function examples that demonstrate a range of functionality, including how to use bitwise and basic aggregate functions.

---

**Note:** The example syntax occasionally wraps due to space constraints. Wrapped lines should be entered on a single line.

---

### **CREATE LIBRARY statement**

---

Identify an external library, and deploy the functions in that library.

This example uses the library file `Functions.class`, which is included with Event Stream Processor. If you are creating a library within ESP Studio using an external file, the `CLASSPATH` variable should contain the library file source directory. If you are not using ESP Studio, you can edit the project configuration file (`.ccr`) to set the Java-classpath option to the library file source directory.

The example begins with the **CREATE LIBRARY** statement, which creates a Java-language library named `SC1` from the `Functions.class` file.

```
CREATE LIBRARY SC1 LANGUAGE java FROM 'Functions' (  
    integer intdiffj(integer, integer);  
    string stringaddj (string, string);  
);
```

The example creates two schemas named `Schema1` and `OutSchema`. The example then creates an input window named `win1` that references `Schema1`, and an output window named `OutWin` that references `OutSchema`. Manually load data into `win1`.

```
CREATE INPUT WINDOW win1 SCHEMA Schema1  
    PRIMARY KEY (fcol5)  
    KEEP ALL  
;  
  
CREATE OUTPUT WINDOW OutWin Schema OutSchema  
    PRIMARY KEY DEDUCED  
AS  
    SELECT a.intcol1,  
        a.intcol2,  
        SC1.intdiffj (a.intcol1, a.intcol2)as library_int,  
        a.fcol5,  
        a.stringcol1,  
        a.stringcol2,  
        SC1.stringaddj(a.stringcol1, a.stringcol2) as library_string
```

```
FROM win1 a
;
```

## Aggregate Functions

---

Apply **first**, **last**, **max**, and **min** functions to outgoing data.

The example creates two schemas named `TradeSchema` and `OpenCloseMinMaxSchema`, and an input window named `TradeWindow`, to which it attaches a File CSV Input adapter.

The example then creates an output window named `OutOpenCloseMinMax`, which uses the structure defined in `OpenCloseMinMaxSchema`. The **SELECT** clause returns the first, last, minimum, and maximum values from the data in `TradeWindow`, and groups the results by `Symbol`.

```
CREATE OUTPUT Window OutOpenCloseMinMax
  SCHEMA OpenCloseMinMaxSchema
  PRIMARY KEY DEDUCED
AS
  SELECT
    TradeWindow.Symbol      as Symbol,
    first(TradeWindow.Price) as OpenPrice,
    last(TradeWindow.Price)  as ClosePrice,
    min(TradeWindow.Price)   as MinPrice,
    max(TradeWindow.Price)   as MaxPrice

  FROM TradeWindow
  GROUP BY TradeWindow.Symbol;
```

## Bitwise Functions

---

Apply **bitand**, **bitor**, **bitshiftright**, **bitshiftleft**, and **bitmask** operations to an output window.

The example creates two schemas named `IntNumbersSchema` and `ResultNumbersSchema`.

The example applies bitwise functions to `ResultNumbersSchema`. Bitwise functions allow you to access and manipulate the individual bits that make up the data.

```
CREATE SCHEMA IntNumbersSchema (
  IntNumber INTEGER
);

CREATE SCHEMA ResultNumbersSchema (
  IntNumber      INTEGER,
  Bit_Shift_Left INTEGER,
  Bit_Shift_Right INTEGER,
```

```

    Bit_Mask          INTEGER,
    Bit_And           INTEGER,
    Bit_Or            INTEGER
);

CREATE Input Window InNumbers
SCHEMA IntNumbersSchema
Primary Key (IntNumber);

CREATE OUTPUT WINDOW OutNumbers
SCHEMA ResultNumbersSchema
PRIMARY KEY ( IntNumber)
AS
SELECT
    i.IntNumber          as IntNumber,
    bitshiftleft(i.IntNumber, 2) as Bit_Shift_Left,
    bitshiftright(i.IntNumber, 2) as Bit_Shift_Right,
    bitmask(0, 4)        as Bit_Mask,
    bitand(i.IntNumber, 4) as Bit_And,
    bitor(i.IntNumber, 4) as Bit_Or
FROM
    InNumbers i;

ATTACH INPUT ADAPTER InAdapter
TYPE dsv_in
TO InNumbers
PROPERTIES
    dir='../exampledata',
    file = 'Numbers1000.csv' ,
    delimiter = ',' ;

```

## Data Aggregation

Read data from a comma-separated value (.csv) file, and aggregate the data using a volume-weighted average price (**vwap**) function.

The example creates a schema named TradeSchema, which is referenced by an input window named TradeWindow. The example attaches a File CSV Input adapter to TradeWindow.

The example creates an output window named VwapWindow, which outputs the results of the volume-weighted average price of the trade values processed by TradeWindow. The results are grouped by Symbol.

```

CREATE output WINDOW VwapWindow
SCHEMA (Symbol STRING, vwap MONEY(4))
PRIMARY KEY DEDUCED
AS
SELECT TradeWindow.Symbol AS Symbol,
    ((SUM(TradeWindow.Price*TradeWindow.Volume)) /
(SUM(TradeWindow.Volume))) AS vwap

```

## CHAPTER 4: Function Examples

```
FROM TradeWindow  
GROUP BY TradeWindow.Symbol;
```



## CHAPTER 5      **Store Examples**

Event Stream Processor includes CCL examples that demonstrate how to create default, memory, and log stores.

---

**Note:** The example syntax occasionally wraps due to space constraints. Wrapped lines should be entered on a single line.

---

### **STORES**

---

Create default, memory, and log stores.

The example creates a memory store named `MemStore`, a default store named `DefaultStore`, and a log store named `LogStore`. Each store retains their default parameter values.

```
CREATE MEMORY STORE MemStore
  PROPERTIES INDEXSIZEHINT = 8 , INDEXTYPE = 'TREE' ;

CREATE DEFAULT MEMORY STORE DefaultStore
  PROPERTIES INDEXSIZEHINT = 8 , INDEXTYPE = 'TREE' ;

CREATE LOG STORE LogStore
  PROPERTIES FILENAME = 'mylog.log' , MAXFILESIZE = 8 ,
    SYNC = FALSE , SWEEPAMOUNT = 20 ,
    RESERVEPCT = 20 , CKCOUNT= 10000 ;
```

The example creates an input window named `TradesWindowMem` that references `MemStore` and an output window named `DefaultStoreWindow` that uses `SELECT all (*)` syntax to retrieve all data columns from `TradesWindowMem`.

The example creates an output window named `LogStoreWindow` that references `LogStore`. `LogStoreWindow` uses **SELECT** and **FROM** clauses to pull timestamp, price, symbol, and volume data from `TradesWindowMem`.

The example attaches a File CSV Input adapter named `InConn` to `TradesWindowMem`.



## CHAPTER 6 Flex Examples

Event Stream Processor includes several Flex examples that demonstrate a range of functionality, including how to use SPLASH syntax, opcodes, timers, **if/then/else** conditions, and event caches.

---

**Note:** The example syntax occasionally wraps due to space constraints. Wrapped lines should be entered on a single line.

---

### Data Management with Flex Streams

---

Use a Flex stream to manage your data.

The example creates three schemas named `TradeSchema`, `Totalschema`, and `Tutelage`, and one input window named `TradeWindow`. The File CSV Input adapter is attached to `TradeWindow`.

The example then creates a Flex stream named `TrackOldTrades` that outputs data from `TradeWindow` to `OldTradeEvents`. The **switch** statement supports only outputs for inserts and updates; as a result, deletes are not passed to the output window.

```
CREATE FLEX TrackOldTrades
  IN TradeWindow
  OUT OUTPUT WINDOW OldTradeEvents
  SCHEMA DeleteOrExpireSchema
    Primary Key (DeleteOrExpireTime, Ts)
BEGIN
  declare
    integer oc;
  end;

  ON TradeWindow {

    oc := getOpcode(TradeWindow);

    switch (oc){
      case insert:
        output [      Ts=TradeWindow.Ts;|
                    Symbol=TradeWindow.Symbol;
                    TotalPrice = TradeWindow.Price * TradeWindow.Volume;
                    Counter =1; ];
        break;
      case update:
        output [      Ts=TradeWindow.Ts;|
                    Symbol=TradeWindow.Symbol;
                    TotalPrice = TradeWindow.Price * TradeWindow.Volume;
                    Counter = 0; ];
    }
  }
END
```

```

        break;
    case delete:
        break;
    Default:
        break;
    } } ;END;
CREATE OUTPUT WINDOW OutWin
Schema Tutelage Primary Key deduced
as
Select o1.Symbol as Symbol,
       Sum(o1.TotalPrice) as TotalPrice,
       Sum(o1.Counter) as Counter
from OutWin1 o1
Group by o1.Symbol
;

```

## Multiple Inputs

---

Use multiple Flex streams with multiple inputs.

The example creates two input windows named Trades 1 and Trades 2.

The example then creates a Flex stream named TradesMSFTFlexStream that joins the two input windows, and adds an output window called TradesMSFTFlexStream.

```

CREATE FLEX Ccl_2_TradesMSFTFlexStream
IN Trades2, Trades1
OUT OUTPUT WINDOW TradesMSFTFlexStream
SCHEMA (Id INTEGER, Symbol STRING, TradeTime DATE, Price FLOAT,
Shares INTEGER, Corr INTEGER)
PRIMARY KEY (Id)
BEGIN
    ON Trades1 {
        if (Trades1.Symbol = 'MSFT') output copyRecord(Trades1);
    };

    ON Trades2 {
        if (Trades2.Symbol = 'MSFT') output copyRecord(Trades2);
    };
END;

```

The example creates another Flex stream (TradesCSCOFlexStream) that joins the Trades1 and Trades2 windows.

```

CREATE FLEX Ccl_4_TradesCSCOFlexStream

IN Trades1, Trades2
OUT OUTPUT WINDOW TradesCSCOFlexStream
SCHEMA (Id INTEGER, Symbol STRING, TradeTime DATE, Price FLOAT,
Shares INTEGER, Corr INTEGER)
PRIMARY KEY (Id)

BEGIN

```

```

ON Trades1 {
if (Trades1.Symbol = 'CSCO') output copyRecord(Trades1);
};

ON Trades2 {
if (Trades2.Symbol = 'CSCO') output copyRecord(Trades2);
};

```

Finally, the example creates a Flex stream named `TradesPickedFlexStream` that joins `TradesMSFTFlexStream` and `TradesCSCOFlexStream`.

```

CREATE FLEX Ccl_5_TradesPickedFlexStream

  IN TradesMSFTFlexStream, TradesCSCOFlexStream
  OUT OUTPUT WINDOW TradesPickedFlexStream
  SCHEMA (Id INTEGER, Symbol STRING, TradeTime DATE, Price FLOAT,
Shares INTEGER, Corr INTEGER)
  PRIMARY KEY (Id)

BEGIN

ON TradesMSFTFlexStream {
if (TradesMSFTFlexStream.Price >= 93) output
copyRecord(TradesMSFTFlexStream);
};

ON TradesCSCOFlexStream {
if (TradesCSCOFlexStream.Price >= 74.5) output
copyRecord(TradesCSCOFlexStream);
};

END;

```

## Average Trade Price with Timer

---

Use a timer to send a new row to an output window every five seconds.

The example creates a schema named `TradesSchema` and an input window named `TradeWindow`. The File CSV Input adapter is attached to the window.

The example creates a Flex stream named `FlexTimer` that places a data retention policy of 10 rows on `TradeWindow`. The **ON** clause tells the project server to apply the computation `vvalue ++` to the trade price every 5 seconds. This expression increments the current value of the local variable `vvalue`.

```

CREATE FLEX FlexTimer IN TradeWindow
  KEEP 10 ROWS
  OUT OUTPUT WINDOW SimpleOutput
  SCHEMA ( a integer, b string)

```

```

        PRIMARY KEY ( a)BEGIN
declare
    integer vvalue := 0;
END;    ON TradeWindow    {    }    ;
every 5 seconds {
    vvalue ++;
    output [a=vvalue; b='msg1'|];
};END;

```

## Variables in the DECLARE Block

---

Define a variable, then use the variable in both a regular stream and Flex stream.

The example specifies a default value of 1000 for the variable `ThresholdValue`.

```

declare
    INTEGER ThresholdValue := 1000;
end;

```

The example creates two schemas named `TradeSchema` and `ControlSchema`. An input window named `TradeWindow` references `TradeSchema`, and an input stream named `ControlMsg` references `ControlSchema`.

The example then creates an output window named `OutTradeWindow`. The **SELECT** clause sends rows greater than `ThresholdValue` to `OutTradeWindow`.

```

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;

```

The example creates a Flex stream named `FlexControlStream` to process the control messages. The **BEGIN** syntax introduces conditions based on control messages. If the control message is set, the `ThresholdValue` is set to equal the control message value instead of the default 1000.

```

CREATE FLEX FlexControlStream
    IN ControlMsg
    OUT OUTPUT WINDOW SimpleOutput
    SCHEMA ( a integer, b string, c integer)
    PRIMARY KEY ( a)
BEGIN
    ON ControlMsg
    {
        if ( ControlMsg.Msg = 'set')
{ThresholdValue:=ControlMsg.Value;}

```

```

        output [a=ControlMsg.Value; b=ControlMsg.Msg;
c=ThresholdValue; |];
    }
;
END
;

```

Finally, the example creates two **ATTACH ADAPTER** instances named `csvInCntMsg` and `csvInConn1` using the File CSV Input adapter. In the first instance, the adapter is attached to `ControlMsg` and assigned to `RunGroup1`. In the second instance, the adapter is attached to `TradeWindow` and assigned to `RunGroup2`. The **ADAPTER START GROUPS** statement tells the project server to read the control messages first, then the stock trades data.

## Event Cache

---

Use an event cache in an output window.

The example creates an input window named `Trades` and an output window named `Last5MinuteStats`.

The examples uses the **DECLARE** block to place an event cache on the `Trades` window. As a result, the `Last5MinuteStats` window retains the last 300 seconds of data for every symbol cached.

```

DECLARE
    eventCache(Trades[Symbol], 300 seconds) stats;
END
AS
    SELECT Trades.Symbol AS symbol,
           max(stats.Price) AS MaxPrice,
           sum(stats.Shares) AS Volume
    FROM Trades
    GROUP BY Trades.Symbol;

```

The example creates an output window named `Last10TradesStats` and uses the **DECLARE** block to place another event cache on the `Trades` window. As a result, the `Last10TradesStats` window retains the last 10 trades for every symbol cached in the `Trades` window.

```

CREATE OUTPUT WINDOW Last10TradesStats
    SCHEMA (
        symbol STRING,
        MaxPrice MONEY(4),
        Volume LONG)
    PRIMARY KEY DEDUCED
DECLARE
    eventCache(Trades[Symbol], 10 events) stats;
END
AS
    SELECT Trades.Symbol AS symbol,

```

```

        max(stats.Price) AS MaxPrice,
        sum(stats.Shares) AS Volume
FROM Trades
GROUP BY Trades.Symbol;

```

## SPLASH with if/then/else

Use a SPLASH **if/then/else** statement and perform the same logic using a **switch** statement.

The example creates a schema called TradeSchema, and an input window called TradeWindow that references the schema. The File CSV Input adapter is attached to the window.

The example then performs a SPLASH **if/then/else** function with nested **if** statements.

```

CREATE FLEX FlexIfThenElse IN TradeWindow
  OUT OUTPUT WINDOW FlexIFOut
  Schema TradeSchema
  Primary Key (Ts)BEGIN   ON TradeWindow   {
    if ( TradeWindow.Price > 100){
      if ( TradeWindow.Price * TradeWindow.Volume < 1000000) {
output (TradeWindow);}
    }

```

These **if** statements tell the project server to output trade data values if the product of TradeWindow.Price \* TradeWindow.Volume is less than 1 million. An **else if** statement executes if the conditions are not true.

```

    Else if ( TradeWindow.Price > 10){
      if ( TradeWindow.Price * TradeWindow.Volume < 10000)
{ output (TradeWindow);}
    }

```

The **else if** statement tells the project server to output trade data values greater than 10 if the total value of shares in the window are less then 10 thousand. An additional **else** statement executes if these conditions are not true.

```

    Else {
      if ( TradeWindow.Price * TradeWindow.Volume < 1000)
{ output (TradeWindow);}
    } } ;END;

```

The **else** statement tells the project server to complete its output when the total value of shares in the window are less than 1000, and the preceding **if/else** conditions are not true.

The example then uses **switch** syntax to achieve the same overall conditions:

```

CREATE FLEX FlexCase IN TradeWindow
  OUT OUTPUT WINDOW FlexCaseOut Schema TradeSchema

```



```

    Primary Key (Ts)
BEGIN
  ON TradeWindow
  {
    switch ( to_integer(log(to_float(TradeWindow.Price)))){
      case 0: // price less than 10
        if ( TradeWindow.Price * TradeWindow.Volume < 1000) {
output (TradeWindow);}
        break;
      case 1: // price between 10 and 100
        if ( TradeWindow.Price * TradeWindow.Volume < 10000) {
output (TradeWindow);}
        break;
      default: // price 100 or bigger
        if ( TradeWindow.Price * TradeWindow.Volume < 1000000)
{ output (TradeWindow);}
        break;
    }
  }
;
END
;

```

The **switch** syntax also converts `TradeWindow.Price` values to float, applies a logarithm to the values, then converts them to integer.

## SPLASH with getOpcode

---

Use a Flex stream to capture items when they are deleted or expire.

The example creates a schema named `TradeSchema`, then another schema named `DeleteOrExpireSchema`, which inherits the structure of `TradeSchema`. The example creates an input window named `TradeWindow`, to which the File CSV Input adapter is attached.

The example then creates a Flex stream named `TrackOldTrades` that outputs data from `TradeWindow` to `OldTradeEvents`.

```

CREATE FLEX TrackOldTrades
  IN TradeWindow
  OUT OUTPUT WINDOW OldTradeEvents
  SCHEMA DeleteOrExpireSchema
    Primary Key (DeleteOrExpireTime, Ts)
BEGIN
  declare
    integer oc;
  end;

```

The **getOpcode** function determines the operation that is performed on the window. The **switch** statement only processes deletes.

## CHAPTER 6: Flex Examples

```
ON TradeWindow
{
    oc := getOpcode(TradeWindow);

    switch (oc){

        case delete:
            output [DeleteOrExpireTime = now();|
                Ts= TradeWindow.Ts; Symbol=TradeWindow.Symbol ;
                Price = TradeWindow.Price; Volume =
TradeWindow.Volume; ];
            break;
        Default:
            break;
    }
}
;
END
;
```

Event Stream Processor includes examples on how to use the **DECLARE** block, including declaring parameters and functions.

---

**Note:** The example syntax occasionally wraps due to space constraints. Wrapped lines should be entered on a single line.

---

## CCL Function

---

Define a function using the **DECLARE** block.

The example creates a schema named `TradeSchema`, then uses the **DECLARE** block to declare the function `MyWeightedAverage`, which includes variables `Value1` and `Value2`. The example also creates the local variable `Weight1`. A series of **if** and **else if** conditions determine the value of `Weight1` based on whether `Value 2` is greater or less than the specified values. The resulting `Weight1` value becomes a parameter in the **to\_money** function.

```
DECLARE Money(2) MyWeightedAverage
  (Money(2) Value1, Integer Value2)
{
  float Weight1 := 1.0;

  IF (Value2 > 10000 )
    { Weight1 := 0.5; }
  ELSE IF (Value2 > 4000)
    {Weight1 := 0.75; }
  ELSE IF (Value2 < 100)
    { Weight1 := 3.0; }
  ELSE IF (Value2 < 500)
    { Weight1 := 0.25; }
  RETURN to_money(Value1 * Weight1 ,2);
}
end;
```

The example creates an input window named `TradeWindow` that references `TradeSchema`, and an output window named `OutWeightedAverage` that specifies an inline schema. `OutWeightedAverage` uses the `MyWeightedAverage` function within the **avg()** function.

```
CREATE OUTPUT WINDOW OutWeightedAverage
  SCHEMA ( Symbol String, avgPrice Money(2), wavgPrice Money(2))
  PRIMARY KEY deduced
```

```

AS
SELECT
    t.Symbol,
    avg(t.Price) avgPrice,
    avg(MyWeightedAverage(t.Price, t.Volume)) wavgPrice
FROM
    TradeWindow t
Group by t.Symbol
;

```

The example concludes by attaching a File CSV Input adapter named `csvInConn1` to `TradeWindow`.

## Parameter Declaration

---

Declare a parameter, then reference it in an output window.

The example declares a parameter called `ThresholdValue` in the DECLARE block, for which it sets the default value 1000. You can change the default value at runtime, or in the project configuration file.

```

DECLARE
    PARAMETER INTEGER ThresholdValue := 1000;
end;

```

The example creates an input window named `TradeWindow` and an output window named `TradeOutWindow`. `TradeOutWindow` uses a **SELECT** statement to pull data from `TradeOptMatch`; a **WHERE** clause tells `TradeOutWindow` to output only data from `TradeWindow` where the product of `TradeWindow.Volume` is greater than the value set for the `ThresholdValue` parameter.

```

CREATE OUTPUT WINDOW TradeOutWindow
    SCHEMA (Ts BIGDATETIME, Symbol STRING, Price MONEY(2), Volume
    INTEGER)
    PRIMARY KEY (Ts)
AS
    SELECT * from TradeWindow WHERE TradeWindow.Volume >
    ThresholdValue;

```

The example attaches a File CSV Input adapter named `csvConn1` to `TradeWindow`.

## CHAPTER 8 Data Selection Examples

Event Stream Processor includes several data selection examples that demonstrate a range of functionality, including how to apply **GROUP BY**, **AGING**, and **WHERE** clauses to data.

---

**Note:** The example syntax occasionally wraps due to space constraints. Wrapped lines should be entered on a single line.

---

### AGING Column

---

Use the **AGING** clause to set an age column for an output window.

The example creates a memory store named `memory1`, followed by an input window named `TradesWindow` that uses the `memory1` store. The example attaches the File CSV Adapter to `TradesWindow`.

```
CREATE MEMORY STORE memory1
  PROPERTIES INDEXTYPE='tree', INDEXSIZEHINT=8;

CREATE INPUT WINDOW TradesWindow
  SCHEMA (
    Ts bigdatetime ,
    Symbol STRING,
    Price MONEY(2),
    Volume INTEGER)
  PRIMARY KEY (Ts)
  STORE memory1;
```

The example creates an output window named `AgingWindow`. The age column for the output window increments every 10 seconds until the age column is equal to 20.

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

## AGING Column with Time Option

---

Use the **AGING** clause to set an age column with time option for an input window.

The example creates a schema named `TradeSchema` and another schema named `TradeAgeSchema`, which inherits the structure of `TradeSchema`. `TradeAgeSchema` also defines three columns named `AgeColumn`, `AgeStartTime`, and `ctime`.

```
Create Schema TradeAgeSchema Inherits TradeSchema
      (AgeColumn integer,
       AgeStartTime bigdatetime, ctime bigdatetime);
```

The example creates an input window named `TradeWindow` that references `TradeSchema`, and an output window named `AgeWindow` that references `TradeAgeSchema`. The example uses the **AGES EVERY** syntax to increment `AgeWindow` every 6 seconds until the age column is equal to 10. A **SELECT** clause places a start time condition on `AgeWindow`, so that the updates specified by the **AGING** clause do not start until 6 minutes after the current time.

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

CREATE OUTPUT WINDOW AgeWindow SCHEMA TradeAgeSchema
  PRIMARY KEY DEDUCED
  AGES EVERY 6 SECONDS
  SET AgeColumn 10 TIMES
  FROM AgeStartTime

AS An
  SELECT * , 1 as AgeColumn,
         now() + 360000000
         as AgeStartTime, now() as ctime
  FROM TradeWindow ;
```

The example then attaches a File CSV Input adapter named `csvInConn1` to `TradeWindow`.

## Data Aggregation

---

Read data from a comma-separated value (`.csv`) file, and aggregate the data using a volume-weighted average price (**vwap**) function.

The example creates a schema named `TradeSchema`, which is referenced by an input window named `TradeWindow`. The example attaches a File CSV Input adapter to `TradeWindow`.

The example creates an output window named `VwapWindow`, which outputs the results of the volume-weighted average price of the trade values processed by `TradeWindow`. The results are grouped by `Symbol`.

```
CREATE output WINDOW VwapWindow
SCHEMA (Symbol STRING, vwap MONEY(4))
PRIMARY KEY DEDUCED
AS
  SELECT TradeWindow.Symbol AS Symbol,
         ((SUM(TradeWindow.Price*TradeWindow.Volume)) /
(SUM(TradeWindow.Volume))) AS vwap
FROM TradeWindow
GROUP BY TradeWindow.Symbol;
```

## Data Aggregation with Filter

---

Use the **HAVING** clause to place a filter on a window.

The example creates an input window named `TradeWindow`, to which it attaches a File CSV Input adapter named `csvInConn1`.

The example creates an output window named `VwapWindow`, which outputs the results of the volume-weighted average price of the trade values processed by `TradeWindow`. The results are grouped by `Symbol`. The **HAVING** clause places a filter condition on `TradeWindow` that tells the project server to publish **vwap** results only when the sum of all `Volume` values for a `Symbol` is greater than 100,000.

```
CREATE OUTPUT WINDOW VwapWindow
SCHEMA (Symbol STRING, vwap MONEY(4))
PRIMARY KEY DEDUCED
AS
  SELECT TradeWindow.Symbol AS Symbol,
         SUM(TradeWindow.Price * TradeWindow.Volume) /
SUM(TradeWindow.Volume) AS vwap
FROM TradeWindow
GROUP BY TradeWindow.Symbol
HAVING
  SUM(TradeWindow.Volume) > 100000;
```

## GROUP BY Clause with last() Function

---

Use the **last** function with **SELECT** clause results. Refer to the results of the **SELECT** clause in a **HAVING** clause.

The example creates a schema named `TradeSchema`.

```
Create Schema TradeSchema
(Ts bigdatettime, Symbol STRING, Price MONEY(4), Volume
INTEGER);
```

## CHAPTER 8: Data Selection Examples

The example creates the schema `TradesWidthDelaySchema`, and uses the **INHERITS** syntax to apply the structure of `TradeSchema` to `TradesWidthDelaySchema` with row delay.

```
CREATE SCHEMA TradesWidthDelaySchema INHERITS TradeSchema
  (RowDelay long);
```

The example creates an input window named `TradeWindow`, to which it attaches the File CSV Input adapter.

The example then creates an output window named `TradesWithDelay` that uses the structure defined in `TradesWidthDelaySchema`. The **SELECT** clause places a row delay on timestamp, symbol, price, and volume data rows. The **HAVING** clause references the `RowDelay` column in the results of the query by not specifying a window name. The **HAVING** clause limits the output window to rows in which the delay is greater than 10 milliseconds.

```
SELECT
  TradeWindow.Ts Ts,
  TradeWindow.Symbol Symbol,
  TradeWindow.Price Price,
  TradeWindow.Volume Volume,
  timeToMsec (TradeWindow.Ts) - timeToMsec (last (TradeWindow.Ts,1))
  as RowDelay
FROM
  TradeWindow
GROUP BY
  TradeWindow.Symbol
Having .RowDelay > 10
;
```

The example creates an output window named `OutTrades` that uses the structure defined in `TradeSchema`. The **GROUP BY** statement processes the selected rows by `Symbol` when the trade price is greater than the last trade price processed. Based on the previous arguments, the project server recognizes when the trade price has increased and the time between trades is greater than 10 milliseconds.

```
GROUP BY
  TradeWindow.Symbol
having
  TradeWindow.Price > last (TradeWindow.Price,1)
;
```



## KEEP Clause

---

Place a **KEEP** clause on an output window.

The example creates an input window named `TradesWindow` and an output window named `KeepCountWindow`. `KeepCountWindow` has a **KEEP** clause that keeps 10 rows at a time in the window.

```
CREATE OUTPUT WINDOW KeepCountWindow
  SCHEMA ( Symbol STRING, Ts bigdatetime )
  PRIMARY KEY (Ts)
  KEEP 10 ROWS
AS
  SELECT TradesWindow.Symbol AS Symbol, TradesWindow.Ts AS Ts
  FROM TradesWindow
;
```

The example attaches a File CSV Input adapter named `InConn` to `TradesWindow`, and a File CSV Output adapter named `OutConn` to `KeepCountWindow`.

## KEEP Clause with AGING Clause

---

Place **KEEP** and **AGING** clauses on an output window.

The example creates a schema named `TradeSchema` and another schema named `TradeAgeSchema` which inherits the structure of `TradeSchema`. `TradeAgeSchema` also defines two columns named `AgeColumn` and `AgeStartTime`.

```
Create Schema TradeAgeSchema Inherits TradeSchema
  (AgeColumn integer,
   AgeStartTime bigdatetime);
```

The example creates an input window named `TradeWindow` that references `TradeSchema`, to which it attaches a File CSV Input adapter.

Finally, the example creates an output window named `KeepAgeWindow` that references `TradeAgeSchema`. `KeepAgeWindow` has a **KEEP** clause that keeps 20 rows in the window at a time. The example also uses the **AGES EVERY** syntax to update `KeepAgeWindow` every 3 seconds until the age column is equal to 10. A **SELECT** clause places a start time condition on `AgeWindow`, so that the updates specified by the **AGING** clause do not start until 6 minutes after the current time.

```
CREATE OUTPUT WINDOW KeepAgeWindow
  SCHEMA TradeAgeSchema
  PRIMARY KEY DEDUCED
  KEEP 20 ROWS
  AGES EVERY 3 SECONDS SET AgeColumn 10 TIMES FROM AgeStartTime
```

```
AS
  SELECT * ,
         1 as AgeColumn,
         now() + 360000000 as AgeStartTime
  FROM TradeWindow ;
```

### KEEP ALL Clause

---

Use the **KEEP ALL** clause with an output window.

The example creates a schema named `TradeSchema`. The example creates an input window named `TradeWindow` that references `TradeSchema`, to which it attaches a File CSV Input adapter.

The example creates an output window named `KeepAllWindows`, which uses the **KEEP ALL** clause to retain all data from `TradeWindow` and group the results by `Symbol`.

```
CREATE OUTPUT WINDOW KeepAllWindows
  SCHEMA (Symbol string, RowCount INTEGER)
  PRIMARY KEY DEDUCED KEEP all
AS
  SELECT TradeWindow.Symbol as Symbol, count(TradeWindow.Symbol) as
  RowCount
  FROM TradeWindow
  group by TradeWindow.Symbol
;
```

### KEEP LAST clause

---

Place a **KEEP LAST** clause on an input window.

The example creates a schema named `TradeSchema` that is referenced by an input window named `TradeWindow`.

The example then creates an output window named `KeepLastWindow` that outputs data from `TradeWindow`. `KeepLastWindow` has a **KEEP** clause that keeps only the last `TradeWindow` row processed by `KeepLastWindow`.

```
CREATE OUTPUT WINDOW KeepLastWindow
  Schema ( Symbol string, RowCount INTEGER)
  PRIMARY KEY DEDUCED KEEP LAST
AS
  SELECT TradeWindow.Symbol as Symbol,
         count(TradeWindow.Symbol) as RowCount
  FROM TradeWindow
  group by TradeWindow.Symbol
;
```

The example concludes by attaching a File CSV Input adapter named `csvInConn1` to `TradeWindow`.

## KEEP PER Clause

---

Use the **KEEP PER** clause with an input window, derived window, or an unnamed window.

The example creates a schema named `TradeSchema`.

```
CREATE SCHEMA TradeSchema (
    Id          long,
    Symbol      STRING,
    Price       MONEY (4),
    Volume      INTEGER,
    TradeTime   DATE
);
```

The example then creates an input window named `Trades` which keeps the last 10,000 rows per `Symbol`.

```
CREATE INPUT WINDOW Trades
SCHEMA TradeSchema
PRIMARY KEY (Id)
KEEP 10000 ROWS PER (Symbol);
```

The example creates an output window named `Last5TradesPerHour` that retains only the last five rows per `Symbol` per hour.

```
CREATE OUTPUT WINDOW Last5TradesPerHour
    PRIMARY KEY DEDUCED
    KEEP 5 ROWS PER (Symbol, TradeHour)
    AS SELECT trw.*, (trw.TradeTime/3600)*3600 TradeHour
    FROM Trades trw;
```

The example creates an output window named `Last50TradesStats` that retains data for the last 50 trades per `Symbol`.

```
CREATE OUTPUT WINDOW Last50TradesStats
    PRIMARY KEY DEDUCED
    AS SELECT trw.Symbol, MAX(trw.Price) MaxPrice,
    MIN(trw.Price) MinPrice, SUM(trw.Volume) Volume
    FROM Trades trw KEEP 50 ROWS PER (Symbol)
    GROUP BY trw.Symbol;
```

The example concludes by attaching the XML Input Adapter to the input window named `Trades` to process the incoming stream data.

```
ATTACH INPUT ADAPTER xmlInConn1
    TYPE xml_in
    TO Trades
    PROPERTIES
        blockSize=1,
        dir='../exampledata',
```

```
file='Trades.xml',
filePattern='*.xml',
safeOps=false,
skipDels=false;
```

## KEEP UNTIL Clause

---

Use a **KEEP UNTIL** clause with a Jumping Window.

The example creates a schema named `TradesSchema` and an input stream named `Trades` that references `TradesSchema`.

```
CREATE SCHEMA TradesSchema (
  Id      integer
  Symbol  string
  Price   float
  Shares  integer
) ;
```

```
CREATE INPUT STREAM Trades
  SCHEMA TradesSchema ;
```

The example then creates a Flex statement named `Until8PM_Flex` that operates on `Trades` and produces an output window named `Until8PM`. The example deletes all previous rows every five seconds, and purges all the data in `Until8PM` at 8:00 PM once a day.

```
CREATE FLEX Until8PM_Flex
IN Trades
OUT OUTPUT WINDOW Until8PM
  SCHEMA TradesSchema
  PRIMARY KEY (Id)
BEGIN
  DECLARE
    date lastPurgeDate;
  END;
  ON Trades {
  };
  EVERY 1 MINUTE {
    if (isnull(lastPurgeDate) or (trunc(sysdate()) >
      lastPurgeDate and hour (sysbigdatetime()) = 20)) {
      for(rec in Until8pm_stream) {
        output setopcodes(rec, delete);
      }
      lastPurgeDate := trunc(sysdate());
    }
  };
END;
```

## Filter with WHERE Clause

---

Use the **WHERE** clause as a filter on an output window.

The example creates an input window named `TradeWindow` and an output window named `TradeOutWindow`.

The **SELECT** clause returns all (\*) data rows from `TradeWindow`. The **WHERE** clause places a filter on the data when the share volume is less than 10,000. As a result, the project server processes all data rows when the `TradeWindow` contains more than 10,000 shares.

```
CREATE OUTPUT WINDOW TradeOutWindow
  SCHEMA (
    Ts BIGDATETIME,
    Symbol STRING,
    Price MONEY(2),
    Volume INTEGER)
  PRIMARY KEY (Ts)
AS
  SELECT * from TradeWindow
  WHERE TradeWindow.Volume > 10000;
```

## MATCHING clause

---

Place a **MATCHING** clause on an output stream.

The example creates a schema named `TradeSchema`, then two input windows named `InTrades` and `InTrades2`, and an output stream named `TradeOut` that each reference `TradeSchema`.

`TradeOut` uses the **MATCHING** clause to retrieve rows that match over a one-second period.

```
CREATE OUTPUT STREAM TradeOut
  SCHEMA TradeSchema
as
  SELECT
    FirstTrade.*
  FROM
    InTrades as FirstTrade,
    InTrades2 as SecondTrade
  MATCHING
    [1 seconds: FirstTrade , SecondTrade ]
  ON
    FirstTrade.Symbol = SecondTrade.Symbol
  ;
```

The example attaches a File CSV Input adapter named `csvInConn1` to `InTrades`, and a File CSV Input adapter named `csvInConn2` to `InTrades2`. The example also attaches a File CSV Output adapter named `csvOut` to `TradeOut` to publish the matching results to a file, since data cannot be viewed in-stream.

## Matching a Sequence of Events

Place **MATCHING** and **WHERE** clauses on output streams to produce a set of sequenced data.

The example creates three schemas: `StocksSchema`, `OptionsSchema`, and `OutSchema`. The example then creates an input window named `InTrades` that references `StocksSchema`; an input window named `InOptions` that references `OptionsSchema`; and two output streams named `TradeOptMatch` and `TradeOptFilter` that both reference `OutSchema`.

`TradeOptMatch` uses the **MATCHING** clause to retrieve rows that match and have the same trade symbol, over a one-second period. `TradeOptFilter` uses a **SELECT** statement to pull data from `TradeOptMatch`; a **WHERE** clause tells `TradeOptFilter` to output data from `TradeOptMatch` only where the product of `0.005 * TradeOptMatch.StockPrice` is greater than the option price.

```
CREATE OUTPUT STREAM TradeOptMatch
    SCHEMA OutSchema
AS
    SELECT
        t.Ts as Ts,
        o.Ts as OptionTs,
        t.Symbol as Symbol,
        t.Price as StockPrice,
        t.Volume as StockVolume,
        o.StockSymbol as StockSymbol,
        o.OptionSymbol as OptionSymbol,
        o.Price as OptionPrice,
        o.Volume as OptionVolume
    FROM
        InTrades as t,
        InOptions as o
        MATCHING
        [1 seconds: t , o ]
        ON
        t.Symbol = o.StockSymbol

CREATE OUTPUT stream TradeOptFilter
    SCHEMA OutSchema
AS
    SELECT * FROM TradeOptMatch
        WHERE 0.005 * TradeOptMatch.StockPrice <
TradeOptMatch.OptionPrice
```

;

The example attaches a File CSV Input adapter named `csvInConn1` to `InTrades`, and a File CSV Input adapter named `csvInConn2` to `InOptions`. The example also attaches a File CSV Output adapter named `outAdapter` to `TradeOptFilter` to publish the filter results to a file, since data cannot be viewed in-stream.

## Matching Non-Events

---

Place a **MATCHING** clause with a not (!) condition on an output stream.

The example creates a schema named `TradeSchema`, then creates an input window named `InTrades` and an output stream named `TradeOut`, both of which reference `TradeSchema`.

`TradeOut` uses `MATCHING not (!)` syntax to retrieve data for stocks that trade twice, but not three times in a 10-millisecond period.

```
CREATE OUTPUT STREAM TradeOut
    SCHEMA TradeSchema
as
    SELECT
        SecondTrade.*
    FROM
        InTrades as FirstTrade,
        InTrades as SecondTrade,
        InTrades as ThirdTrade
    MATCHING
        [10 milliseconds: FirstTrade , SecondTrade, !ThirdTrade ]
    ON
        FirstTrade.Symbol = SecondTrade.Symbol = ThirdTrade.Symbol
;
```

The example attaches a File CSV Input adapter named `csvInConn1` to `InTrades`. The example also attaches a File CSV Output adapter named `csvOut` to `TradeOut` to publish the matching results to a file, since data cannot be viewed in-stream.

## Row Time

---

Use the `bigdatetime` system column to retrieve row-insertion times.

The example creates a schema named `TradeSchema`.

The example creates the schema `TradesWidthDelaySchema`, and uses the **INHERITS** syntax to apply the structure of `TradeSchema` to `TradesWidthDelaySchema` with row delay.

## CHAPTER 8: Data Selection Examples

The example creates an input window named `TradeWindow`, to which it attaches the File CSV Input adapter.

The example then creates an output window named `TradesWithDelay` that uses the structure defined in `TradesWidthDelaySchema`. The **SELECT** clause places a row delay on timestamp, symbol, price, and volume data rows. The row delay is defined as 10 milliseconds in the **HAVING** clause. The results are grouped by `Symbol`.

```
CREATE OUTPUT WINDOW TradesWithDelay SCHEMA TradesWidthDelaySchema
Primary Key deduced
as
SELECT
    TradeWindow.Ts Ts,
    TradeWindow.Symbol Symbol,
    TradeWindow.Price Price,
    TradeWindow.Volume Volume,
    timeToMsec(TradeWindow.BIGROWTIME ) - timeToMsec(TradeWindow.Ts)
    as RowDelay
FROM
    TradeWindow
GROUP BY
    TradeWindow.Symbol
;
```

## AUTOGENERATE Clause

---

Use the **AUTOGENERATE** clause in an input window to automatically generate values that function as primary keys for input data that does not have a natural primary key.

**Note:** Do not use the **AUTOGENERATE** with upserts. This might produce duplicate rows in a window, especially when the automatically-generated column is a primary key.

---

The example creates a schema named `TradeSchema`.

```
CREATE SCHEMA TradeSchema (
    AutoGenId long,
    Symbol STRING,
    Price MONEY (4),
    Volume INTEGER,
    TradeTime DATE
);
```

The example then creates an input window named `Trades` that uses the schema `TradeSchema` and sets the `AutoGenId` as the primary key. Finally, the example uses the **AUTOGENERATE** clause to automatically generate values for the `AutoGenId` column.

```
CREATE INPUT WINDOW Trades
SCHEMA TradeSchema
PRIMARY KEY (AutoGenId)
AUTOGENERATE (AutoGenId);
```



The example concludes by attaching the XML Input Adapter to the input window named Trades to process the incoming stream data.

```
ATTACH INPUT ADAPTER xmlInConn1
  TYPE xml_in
  TO Trades
  PROPERTIES
    blockSize=1,
    dir='../exampledata',
    file='Trades.xml',
    filePattern='*.xml',
    safeOps=false,
    skipDels=false;
```



Event Steam Processor includes examples for creating and loading modules.

---

**Note:** The example syntax occasionally wraps due to space constraints. Wrapped lines should be entered on a single line.

---

## CREATE MODULE

---

Create a module that can be added to a project later using the **LOAD MODULE** statement.

The example creates a module named `Module1`, identifying the input and output windows that are later defined in the **BEGIN-END** block.

```
CREATE MODULE Module1 IN rawStockFeed OUT infoByStockSymbol
```

In the **BEGIN-END** block, the example declares the parameter `myparam`, for which it sets a default value of 2. The example also creates a memory store named `store1`.

```
BEGIN
  DECLARE
    parameter integer myparam := 2;
  END;

  CREATE DEFAULT MEMORY STORE store1;
```

The example creates two schemas named `inputSchema` and `outputSchema`. It then creates an input window named `rawStockFeed`, which references `inputSchema`, and an output window named `infoByStockSymbol`, which references `outputSchema`. The function `getRecordCount()`, which is referenced later in the statement, is declared using a **DECLARE** block.

The output window `infoByStockSymbol` uses **SELECT** and **FROM** clauses to pull data from `rawStockFeed`. A **WHERE** clause places a filter on the data when the share volume is greater than the value set for `myparam`. The example concludes by closing the **BEGIN-END** block.

```
CREATE OUTPUT WINDOW infoByStockSymbol
SCHEMA outputSchema
  PRIMARY KEY DEDUCED
  DECLARE
    integer recordCount:=1;
    integer getRecordCount() {
      return recordCount++ ;
    }
  END
AS
```

```
SELECT rawStockFeed.Symbol,
       avg(rawStockFeed.Price) AvgPrice,
       sum(rawStockFeed.Volume) Volume,
       count(rawStockFeed.Symbol) NumRecordsForSymbol,
       getRecordCount() TotalNumRecords,
       myparam as dummy
FROM rawStockFeed
WHERE rawStockFeed.Volume > myparam
GROUP BY rawStockFeed.Symbol;
END;
```

### Load Module

---

Import and load a module.

This example uses the **IMPORT** statement to load the module defined in the **CREATE MODULE** example, which is saved as `module1.ccl`.

The example loads `module1.ccl` using the **IMPORT** statement.

```
IMPORT 'module1.ccl';
```

The example creates two schemas named `StocksSchema` and `ComputedStocksSchema`, a default store named `MyStore1`, and a memory store named `MyStore2`.

The example then creates an input window named `InStocks` that references `StocksSchema`, and to which it attaches a File CSV Input adapter named `csvInStocks`.

The example uses the **LOAD MODULE** statement to load `Module1`, linking the input window identified within the module to `InStocks`, and referencing `MyStore1`. This example does not create a new output window, but assigns a new name (`CompStocks2`) to the window loaded from `Module1`. The example also sets a value for the `myparam` parameter declared in `Module1`.

```
LOAD MODULE Module1 AS Module1_instance_01
  IN rawStockFeed = InStocks
  OUT infoByStockSymbol = CompStocks2
  Parameters myparam = 1000
  STORES store1=MyStore1;
```

The example creates an output window named `myw2` that references `ComputedStocksSchema`. **SELECT all (\*)** syntax outputs all data processed by `CompStocks2` to `myw2`.

## CHAPTER 10    **Advanced Examples**

Event Stream Processor includes advanced programming examples that incorporate a variety of CCL elements.

---

**Note:** The example syntax occasionally wraps due to space constraints. Wrapped lines should be entered on a single line.

---

### **Portfolio Valuation**

---

Compute volume-weighted average prices on a stock portfolio.

The example creates an input window named `PriceFeed` and an output window named `VWAP`. `VWAP` outputs the results of the volume-weighted average price of the trade values processed by `PriceFeed`. The results are grouped by `Symbol`. The **cast** function converts share values to float.

```
CREATE OUTPUT WINDOW VWAP
SCHEMA (Symbol STRING, LastPrice FLOAT, VWAP FLOAT, LastTime DATE)
    PRIMARY KEY DEDUCED AS
    SELECT PriceFeed.Symbol AS Symbol,
           PriceFeed.Price AS LastPrice,
           (sum((PriceFeed.Price * cast(FLOAT ,PriceFeed.Shares))) /
            cast(FLOAT ,sum(PriceFeed.Shares))) AS VWAP,
           PriceFeed.TradeTime AS LastTime
FROM PriceFeed
GROUP BY PriceFeed.Symbol;
```

The example creates an input window named `Positions` and an output window named `IndividualPositions`. `IndividualPositions` creates a join between `Positions` and `VWAP` using their symbol values.

```
CREATE OUTPUT WINDOW IndividualPositions
SCHEMA (BookId STRING, Symbol STRING, CurrentPosition FLOAT,
AveragePosition FLOAT)
    PRIMARY KEY (BookId, Symbol) AS
    SELECT Positions.BookId AS BookId, Positions.Symbol AS
Symbol,
           (VWAP.LastPrice * cast(FLOAT ,Positions.SharesHeld)) AS
CurrentPosition,
           (VWAP.VWAP * cast(FLOAT ,Positions.SharesHeld)) AS
AveragePosition
    FROM Positions JOIN VWAP
    ON Positions.Symbol = VWAP.Symbol;
```

## CHAPTER 10: Advanced Examples

The example creates an output window named `ValueByBook`, which uses **SELECT** and **FROM** clauses to pull data from `IndividualPositions` using book ID values. `ValueByBook` groups the data by book ID.

```
CREATE OUTPUT WINDOW ValueByBook
  SCHEMA (BookId STRING, CurrentPosition FLOAT, AveragePosition
  FLOAT)
  PRIMARY KEY DEDUCED AS
  SELECT IndividualPositions.BookId AS BookId,
         sum(IndividualPositions.CurrentPosition) AS CurrentPosition,
         sum(IndividualPositions.AveragePosition) AS AveragePosition
  FROM IndividualPositions
  GROUP BY IndividualPositions.BookId;
```

The example concludes by attaching a File XML Input adapter named `Adapter1` to `PriceFeed`, and another File XML Input adapter named `Adapter2` to `Positions`.

## Trades Log

---

Use a Flex stream to manually delete data from windows.

The example creates a **MEMORY** store named `store1`, then two input windows named `Trades` and `Trades_truncate` that reference `store1`.

The example attaches a File CSV Input adapter named `Adapter1` to `Trades`. The adapter reads sample data from the file `pstrades1.xml` in the `exampledata` folder, and publishes the information to `Trades`.

```
ATTACH INPUT ADAPTER Adapter1
  TYPE xml_in TO Trades
  PROPERTIES
    dir = '../exampledata' ,
    file = 'pstrades1.xml' ;
```

The example creates a Flex statement named `Ccl_2_Trades_log` that operates on `Trades` and `Trades_truncate`, producing an output window named `Trades_log`. Using a **DECLARE** block within the Flex statement, the example declares two longs to store the lowest and the highest sequence number produced in the example so far.

```
CREATE FLEX Ccl_2_Trades_log
  IN Trades, Trades_truncate
  OUT OUTPUT WINDOW Trades_log
  SCHEMA (SequenceNumber LONG, GDOpcode INTEGER, Id INTEGER,
  Symbol STRING, TradeTime DATE, Shares INTEGER, Price
  MONEY(4))
  PRIMARY KEY (SequenceNumber)
  STORE store1
BEGIN
DECLARE
```

```

LONG low;
LONG high;

END;
```

An **ON** clause executes the code below anytime a record comes through on the `Trades` window. A series of **if**, **else**, and **while** conditions tell the project server that, if this is the first record being seen by the Flex stream, it should initialize the high and low sequence numbers. The example uses an iterator to scan all of the records in the `Trades_log` to find the lowest and highest sequence numbers stored in the log. Once the example has finished iterating through `Trades_log`, the highest sequence number that exists in the log and the lowest sequence number are stored, and the iterator is deleted.

```

ON Trades {
  {
    LONG sn;
    /* on the first record, initialize the low, high record
    numbers */
    if (isnull(high))
      {
        for ( Trades_log in Trades_log_stream )
          {
            if (isnull (high))
              {
                high := 0; low := 9223372036854775807;
              }
            sn := Trades_log.SequenceNumber;
            if (sn > high)
              {
                high := sn;
              }
            if (sn < low)
              {
                low := sn;
              }
          }
        /* If high is still null there no records in log stream
        */
        if (isnull(high))
          {
            high := -1; low := 0;
          }
      }
    /* output the incoming record with a record number + opcode
    prepended */
    high := high + 1
    output [SequenceNumber = high; |
    GDOpcode = getOpcode(Trades);
    Id=Trades.Id;
    Symbol=Trades.Symbol;
    TradeTime=Trades.TradeTime;
    Shares=Trades.Shares;
    Price=Trades.Price;
    ];
  }
}
```

```
};
```

The example increments the highest sequence number by 1, and assigns this sequence number to the current trade it is processing. For the first record, the sequence number is 0.

```
high:=(high+ cast(LONG ,1));
    output [SequenceNumber=high; |GDOPcode=getOpCode(Trades);
           Id=Trades.Id; Symbol=Trades.Symbol;
           TradeTime=Trades.TradeTime; Shares=Trades.Shares;
           Price=Trades.Price; ];
    }
};
```

An **ON** clause executes this code anytime a record comes through on the `Trades_truncate` window:

```
ON Trades_truncate {
    {
        LONG i;
        [LONG SequenceNumber; |INTEGER GDOPcode; INTEGER Id;
         STRING Symbol; DATE TradeTime; INTEGER Shares; MONEY(4)
         Price; ] outrec;
```

A series of **if** and **while** conditions provides the format for output. The example gets the sequence number that was provided on `Trades_truncate`. All records with sequence numbers lower than this number are removed from the trades log. If the sequence number requested is larger than or equal to the largest sequence number in the trades log, the example removes all but the latest record from the trades log.

```
i:=Trades_truncate.SequenceNumber;
    if ((high> cast(LONG ,0)))
    {
        if ((i>= high))
            i:=(high- cast(LONG ,1));
        if ((low<= i) and (i< high))
        {
            while ((low<= i))
            {
```

The example creates a record with an opcode of 13 (**SAFE DELETE**) for each sequence number lower than the value provided. Safe delete means the record is deleted from all subsequent windows if it exists; no error occurs if it does not exist.

```
GDOPcode=cast(INTEGER ,null);
    outrec:=[SequenceNumber=low; |
            Id=cast(INTEGER ,null);
            Symbol=cast(STRING ,null);
            TradeTime=cast(DATE ,null);
            Shares=cast(INTEGER ,null);
```



```

        Price=cast(MONEY(4),null); ];
        setOpcode(outrec,13);
        output outrec;
        low:=(low+ cast(LONG ,1));
    }
}
};
END;

```

## Vectors and Dictionaries

Using a vector and dictionary data structure in SPLASH.

This example implements an OUTPUT AFTER logic that accumulates N Trades per Symbol before the rows are outputted for further processing. A data structure combining a dictionary and a vector caches the rows for every symbol until there is at least N rows for a Symbol. N is controlled by the parameter NoOfRows.

To test this model, run it, view the DelayedTrades stream in the stream viewer and manually load input into the Trades stream. You will see rows in the stream viewer only after you insert N trades for a symbol.

```

DECLARE
    integer NoOfRows := 3;
END;

CREATE SCHEMA TradeSchema
    (Id long, Symbol STRING, Price MONEY(4), Volume INTEGER,
    TradeTime DATE);

CREATE INPUT STREAM Trades SCHEMA TradeSchema;

CREATE FLEX DelayedTrades_Flex
IN Trades
OUT OUTPUT STREAM DelayedTrades
    SCHEMA TradeSchema
BEGIN
    DECLARE
        /*Data structure combining a dictionary and a vector
        dictionary(string, vector(typeof(Trades))) cache;
    END;
    ON Trades {
        /*Get the reference to the vector associated with a Symbol
        from the cache.*/
        vector(typeof(Trades)) symbolTrades := cache[Trades.Symbol];
        if(isnull(symbolTrades)){
            /*Create a new vector for this symbol.
            Note that you have to use a new to create a vector or

```

```

dictionary if it is not directly defined in a global or
local declare/end block. In this example the cache
dictionary does not have to be newed because it is
directly defined in the local declare/end block but the
vector inside the dictionary is not.*/
symbolTrades := new vector(typeof(Trades));

//Add the current row to the vector.
push_back(symbolTrades, Trades);

//Assign the vector to the cache for the current Symbol.
cache[Trades.Symbol] := symbolTrades;

exit;
} else {
/*There is a vector already available for the Symbol, so
insert the current row. Note that you don't have to
assign the vector back into the dictionary because the
vector symbolTrades is a reference to the corresponding
vector in the dictionary.*/
push_back(symbolTrades, Trades);
}

//The vector has reached size N.
if(size(symbolTrades) = NoOfRows) {
//Iterate through the rows and output them.
for(rec in symbolTrades) {
output rec;
}

//Prepare for the next N Rows. Clear the vector.
resize(symbolTrades, 0);
}
};
END;

```

## Dictionary of Dictionaries

This example uses one dictionary to sort incoming data for entry into other dictionaries.

This sample shows a flex stream that uses a dictionary to index other dictionaries. The input data comes from a CSV file, `ticks.csv`. The records are inserted into an input window named `TickIn` using a CSV File Adapter. Each record describes a stock sale: it includes the number of shares sold, the symbol of the stock, and the sector of the market (e.g. software) to which the stock belongs. The first dictionary uses the market sector to select (or create) a dictionary and that dictionary uses the symbol to store the record in a cache variable. Every 60 seconds the contents of the cache are sent to the output stream, and removed from the cache.

To check the content of the cache,

1. Enter `esp_client -p localhost:9786/default/project_name -c studio:studio` at the system prompt.

2. Enter `trace_mode` on at the `esp_client>` prompt.
3. Enter `ex `var` `ostream` `cache`` at the `esp_client>` prompt.

After sixty seconds the data are sent to the output stream and removed from the cache.

```
CREATE INPUT WINDOW TickIn SCHEMA (seqnum integer,sector
string,symbol string,volume integer)
PRIMARY KEY(seqnum);

CREATE FLEX flex1
IN TickIn
OUT OUTPUT STREAM ostream SCHEMA (sector string,symbol
string,volume integer)
BEGIN
DECLARE
typedef[string sector; string symbol; | integer volume] outrec;

dictionary(string,dictionary(string,integer)) cache;
END;
ON TickIn
{
    /* Storing data in a dictionary */
    dictionary(string,integer) symcache;
    integer prevTotalVolume := 0;
    symcache:=cache[TickIn.sector];
    /* Check to see if there is a dictionary assigned for the sector,
if not create a new
    * dictionary
    */
    if (isnull(symcache))
    {
        symcache := new dictionary(string,integer);
        cache[TickIn.sector]:=symcache;
    } else {
        prevTotalVolume := symcache[TickIn.symbol];
    }
    symcache[TickIn.symbol]:= prevTotalVolume + TickIn.volume;
};
EVERY 60 SECONDS
{
    /* Displaying the content of the dictionary */
    for (sector_key in cache)
    {
        dictionary(string,integer) symcache;
        /* Here symcache is a reference variable so deleting or
modifying the
        * content of symcache will modify the actual dictionary
content
        */
        symcache:=cache[sector_key];
        for(symbol_key in cache[sector_key])
        {

            outrec rec :=[
            sector = sector_key;
```

## CHAPTER 10: Advanced Examples

```
        symbol = symbol_key;
        volume = symcache[symbol_key];

    ];
    output setOpcode(rec,insert);
}

/* Deleting the content of the dictionary */
for(sector_key in cache)
{
    clear(cache[sector_key]);
}
clear(cache);
};
END;

ATTACH INPUT ADAPTER csvInConn1
TYPE dsv_in
TO TickIn
PROPERTIES
    blockSize=1,
    dateFormat='%Y/%m/%d %H:%M:%S',
    delimiter=',',
    dir='./exampledata',
    expectStreamNameOpcode=false,
    fieldCount=0,
    file='tick.csv',
    filePattern='*.csv',
    hasHeader=false,
    safeOps=false,
    skipDels=false,
    Pollperiod=2,
    timestampFormat='%Y/%m/%d %H:%M:%S';
```

# Index

## A

- adapter examples
  - adapter data with opcodes 5
  - ADAPTER START GROUPS statement 4
  - ATTACH ADAPTER statement 3
  - Database Input Adapter 7
  - Database Input Adapter with polling 9
  - Database Output Adapter 8
  - File CSV Output adapter 6
  - schema inheritance 4
- advanced examples
  - dictionaries 59
  - portfolio valuation using vwap() 55
  - SPLASH 59
  - trades log 56
  - vectors 59
- auto generate 50

## D

- data selection examples 39, 45, 46, 50
  - AGING column 39
  - AGING column with time option 40
  - data aggregation with filter 41
  - filter with WHERE clause 47
  - GROUP BY clause with last() function 41
  - KEEP ALL clause 44
  - KEEP clause 43
  - KEEP clause with AGING clause 43
  - KEEP LAST clause 44
  - MATCHING clause 47
  - matching non-events 49
  - matching sequences of events 48
  - row time retrieval 49
- data structures 60
- DECLARE block examples
  - declaring a function 37
  - parameter declaration 38
- dictionary 60

## F

- Flex examples
  - average trade price with timer 31

- data management with Flex streams 29
- event cache 33
- multiple streams and inputs 30
- SPLASH with getOpcode 35
- SPLASH with if/then/else 34
- variables in the DECLARE BLOCK 32
- function examples
  - aggregate functions 24
  - bitand() 24
  - bitmask() 24
  - bitor() 24
  - bitshiftleft() 24
  - bitshiftright() 24
  - bitwise functions 24
  - CREATE LIBRARY statement 23
  - data aggregation with vwap() 25, 40
  - first() 24
  - last() 24
  - max() 24
  - min() 24

## J

- jumping window 19

## K

- KEEP PER clause 45
- KEEP UNTIL Clause 46

## M

- module examples
  - CREATE MODULE 53
  - load module 54

## P

- parameters 38

## S

- SPLASH 60

## Index

- store examples
  - default, memory, and log stores 27
- stream and window examples
  - delta stream 14
  - input and local streams 13
  - join streams 16
  - join windows 15
  - local windows and output windows 14
  - outer join 17

- stream splitting 20
- union streams 18

## W

- window
  - jumping 19