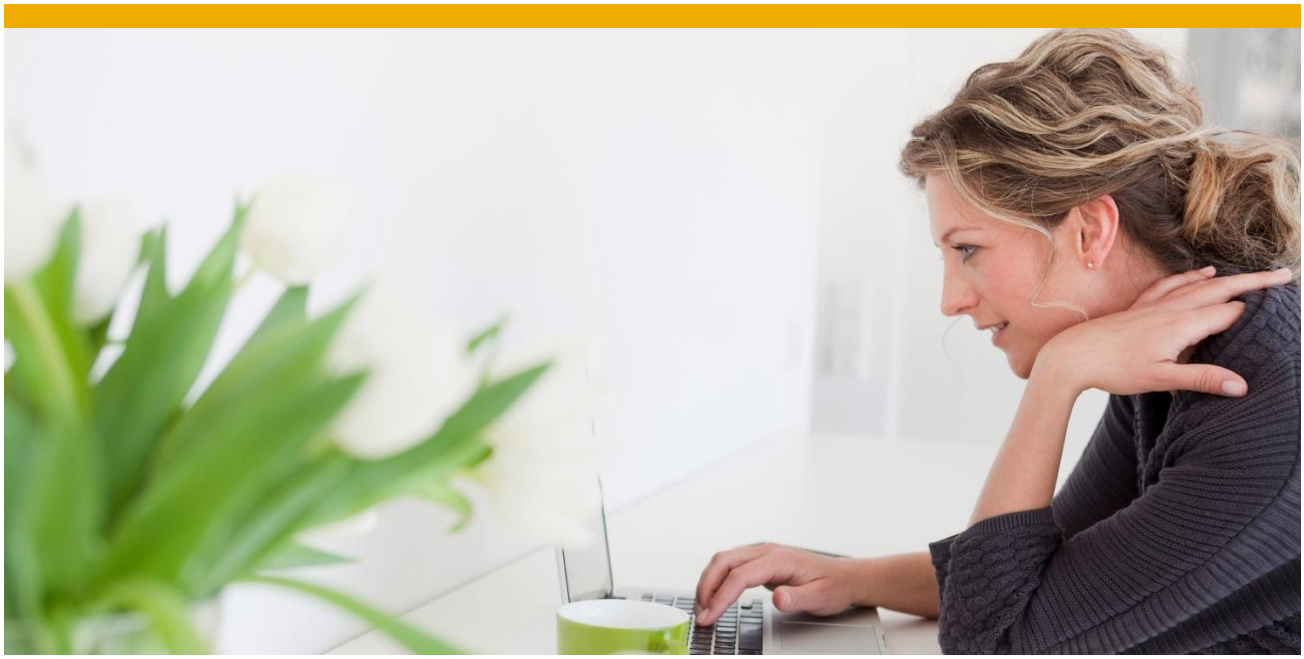


SAP Sybase Event Stream Processor- Design Patterns: CCL and SPLASH (1st edition)

By Vijaigeetha Chittaranjan (Geetha) and David Smith

TABLE OF CONTENTS

INTRODUCTION

The SAP Sybase® Event Stream Processor (ESP) enables you to create and run your own complex event processing (CEP) applications to derive continuous intelligence from streaming event data in real time. CEP applications are written using the Continuous Computation Language (CCL) and the SPLASH scripting language.

AUDIENCE

This paper is intended for use by programmers responsible for coding applications using CCL and SPLASH.

PURPOSE OF THIS DOCUMENT

This document presents examples of CCL patterns and algorithms used to solve common CCL coding problems. The examples shown in this paper are minimally tested solutions that were developed during the coding process of several large projects.

SCOPE OF THIS DOCUMENT

Since there are many different approaches to resolving coding problems, this document is not meant to be a comprehensive coding guide or a replacement to other reference guides. This document is intended as a supplement to those resources and is limited to the approaches used by the authors in other projects. For more details on CCL programming, please see:

http://infocenter.sybase.com/help/topic/com.sybase.infocenter.dc01612.0513/doc/html/title.html

http://infocenter.sybase.com/help/topic/com.sybase.infocenter.dc01621.0513/doc/html/title.html

1: HOW TO WRITE A CCL EQUIVALENT FOR A SQL-JOIN-AGAINST-DATE RANGES

Problem: Create a CCL equivalent to a SQL Join against date ranges.

Consider the following two tables:

1. Transactions by date with their foreign currency amount):

Date	Amount
1/2/2009	1500
2/4/2009	2300
3/15/2009	300
4/17/2009	2200

2. Exchange Rates by date: In this case, the rate for converting a foreign currency amount to a primary currency value (US Dollars):

Date	Rate
2/1/2009	40.1
3/1/2009	41.0
4/1/2009	38.5
5/1/2009	42.7

Consider this workflow:

1. Exchange rates can be entered for arbitrary dates; users can enter them on a daily basis, weekly basis, monthly basis, or at irregular intervals):

2. In order to translate the foreign amount into US dollars, we need to respect the following business rules when coding the application:

    a. If possible, use the most recent previous rate. Therefore, a transaction on 2/4/2009 would use the rate for 2/1/2009, and a transaction on 3/15/2009 would use the rate for 3/1/2009.

    b. If there isn't a rate defined for a previous date, use the most recent rate available. Therefore, a transaction on 1/2/2009 would use the rate for 2/1/2009, since this is the most recent previous rate defined.

Solution: Consider this coding sample:

```
CREATE INPUT STREAM T_Transaction
SCHEMA (Tran_Date date,Amount integer);

CREATE INPUT  STREAM T_ExchangeRates
SCHEMA(Exch_Date date,Rate float);

/*Convert date to long value as long values are easier for computation*/
CREATE OUTPUT STREAM str_Transaction
SCHEMA(Tran_Date date,val long,Amount integer)
AS
SELECT
st1.Tran_Date as Tran_Date,
to_long(st1.Tran_Date) as val,
st1.Amount as Amount
FROM T_Transaction st1;

/*Convert date to long value. Exchange Rates should be in a window*/
CREATE OUTPUT WINDOW W_ExchangeRates
SCHEMA(Exch_Date date,val long,Rate float)
PRIMARY KEY DEDUCED
AS
SELECT
a.Exch_Date as Exch_Date,
to_long(a.Exch_Date) as val,
a.Rate as Rate
FROM T_ExchangeRates a
GROUP BY a.Exch_Date
;

/*
 * Create a Cartesian product stream-window join every transaction will be joined with
 * all rows in the exchange rate window
 */
CREATE OUTPUT STREAM join_out
SCHEMA(Tran_Date date,Exch_Date date,w1_val long,str_val long,Amount integer,Rate float)
AS
SELECT
str.Tran_Date as Tran_Date,
w1.Exch_Date as Exch_Date,
w1.val as w1_val,
str.val as str_val,
str.Amount as Amount,
w1.Rate as Rate
FROM str_Transaction str,W_ExchangeRates w1;

/*
 * Create a output stream that filters out the date values that are < or rather
 * we need only date values that are less than the Transaction date
 */
```

```
CREATE OUTPUT STREAM filter_result
SCHEMA(Tran_Date date,Exch_Date date,w1_val long,str_val long,Amount integer,Rate float)
AS
SELECT * FROM join_out
WHERE join_out.w1_val < join_out.str_val;

/*
 * This is the final result window where the output is grouped based on
 * transaction date and order by exchange date and the last in the group
 * is retrieved.
 */

CREATE OUTPUT WINDOW  result_window
SCHEMA (Tran_Date date,Exch_Date date,w1_val long,str_val long,Amount integer,Rate float)
PRIMARY KEY DEDUCED
AS
SELECT
last(a.Tran_Date) as Tran_Date,
last(a.Exch_Date) as Exch_Date,
last(a.w1_val) as w1_val,
a.str_val as str_val,
last(a.Amount) as Amount,
last(a.Rate) as Rate
FROM filter_result a
GROUP BY a.str_val
GROUP ORDER BY a.w1_val;
```

2: HOW TO CAPTURE ROWS WITH DIFFERENT OPCODES IN SEPARATE STREAMS

Problem: Historically, capturing different opcode data was done using flex operators to produce derived streams with SPLASH. However, using flex operators with SPLASH in this situation often results in an overly complex solution since it requires using three flex operators. There is a much easier way to do this using ESP CCL.

Solution: Below, is an example of a simpler way to capture different opcodes in different streams using CCL:

```
//Input for which we want to capture the different opcodes
CREATE INPUT WINDOW stock
SCHEMA (symbol string,volume integer, price float)
PRIMARY KEY (symbol)
KEEP 10 rows
;
//Capture expired rows using matching clause and check the incoming opcode
CREATE OUTPUT STREAM stream_delete
SCHEMA(symbol string, volume integer, price float)
AS
SELECT *
FROM stock matching[1 second:stock]
ON getOpcode(stock)=delete;

//Capture only updated rows using matching clause and check the incoming opcode
CREATE OUTPUT STREAM stream_update
SCHEMA(symbol string, volume integer, price float)
AS
SELECT *
FROM stock matching[1 second:stock]
ON getOpcode(stock)=update;

//Capture only inserted rows using matching clause and check the incoming opcode.
CREATE OUTPUT STREAM stream_insert
SCHEMA(symbol string, volume integer, price float)
```

```
AS
SELECT *
FROM stock matching[1 second:stock]
ON getOpcode(stock)=insert;
```

3: HOW TO CHANGE THE TIME-BASED POLICY OF A WINDOW DURING RUN TIME

Problem: Currently, there is no direct way to change the time-based policy of a window during runtime using CCL. However, the CCL aging clause can be used to work around this problem. The aging clause can be used to age rows in the window at different times based on a field in the schema. This field should have a different ageStarttime based on the length of time it should remain in the window. Below, is a short example that can be modified as needed. The trick is to include the time period as part of the data.

Solution: Consider the following coding sample for changing the time-based policy of a window:

```
CREATE INPUT STREAM S1
SCHEMA (symbol string, volume integer, price integer, VariableKEEP bigdatetime);

CREATE OUTPUT WINDOW W1
SCHEMA(symbol string, volume integer, price integer, AgeColumn integer, agingTimeField
bigdatetime)
PRIMARY KEY DEDUCED
AGES every 1 second set AgeColumn
FROM agingTimeField
AS SELECT
S1.symbol as symbol,
S1.volume as volume,
S1.price as price,
1 as AgeColumn,
S1.VariableKEEP as agingTimeField
FROM S1
GROUP BY S1.symbol;

CREATE OUTPUT STREAM stream_update
SCHEMA(symbol string,volume integer, price integer)
AS SELECT
W1.symbol as symbol,
W1.volume as volume,
W1.price as price
FROM W1 matching[1 second:W1]
ON getOpcode(W1)=update;
```

Note: If the window retention policy is five days, then the VariableKEEP field should have a value of five days from the present date. If the window retention policy is seven days, delete the rows in the window, and insert the rows with a VariableKEEP field value of seven days from the present date. When the retention period expires, the field AgeColumn will be updated with a value of “1” and the pattern matching query will filter the update, or it could be made to filter a “1”.

4: HOW TO FLATTEN AN ADJACENT (PARENT-TO-CHILD) HIERARCHY IN CCL

Problem: It is often useful to flatten multiple rows of an adjacent hierarchy table into a single row. There are several ways to do this in ESP using CCL. The following section presents two methods for doing this. The first method is a step-by-step approach which is easy to understand. The second method is a query approach that has a complicated join statement.

Consider the following:

1. A table of adjacent hierarchical data (parent-to-child):

Id	Parent id
BSP1	GSP1
GSP1	GSPGROUP
GSPGROUP	NULL
LV10	SS2
SS2	PSS1
PSS1	BSP1

2. A flattened representation of this data:

Id	Parent_id	G_Parent_id	GG_Parentid	GGG_Parentid	GGGG_Parentid
BSP1	GSP1	GSPGROUP	NULL	NULL	NULL
LV10	SSS2	PSS1	BSP1	GSP1	GSPGROUP

Solution: Consider the following methods:

Method 1:

```
CREATE INPUT WINDOW GroupEntity
SCHEMA(Id string,ParentId string,Type_t string, LossFactor float,WarningThreshold
float,LimitThreshold float)
PRIMARY KEY(Id);

/*
 * This is the step by step approach to resolve the Flattening hierarchy issue
 */
//The query does a self-join for the first hierarchy which is id,parent, grandparent
CREATE OUTPUT WINDOW Hierarchy1
SCHEMA(Id string,ParentId string,GParentId string)
PRIMARY KEY(Id)
AS
SELECT
G.Id as Id,
G.ParentId as ParentId,
G2.ParentId as GParentId
From GroupEntity G left join GroupEntity G2
ON G.ParentId=G2.Id ;

/*
 * The query does a self-join for the second hierarchy which is id, parent, grandparent,
 * great-grandparent
 * The output of the previous query Hierarchy1 is the input to Hierarchy2
 */
CREATE OUTPUT WINDOW Hierarchy2
SCHEMA(Id string, ParentId string, GParentId string, GGParentId string)
PRIMARY KEY(Id)
AS
SELECT
G.Id as Id,
G.ParentId as ParentId,
G.GParentId as GParentId,
G2.ParentId as GGParentId
```

```
FROM Hierarchy1 G left join Hierarchy1 G2
ON G.GParentId=G2.Id ;

/*
 * The output of Hierarchy2 is used as the input of Hierarchy3, the query does a
 * self-join for the third hierarchy
 */
CREATE OUTPUT WINDOW Hierarchy3 SCHEMA(Id string,ParentId string,GParentId
string,GGParentId string,GGGParentId string)
PRIMARY KEY(Id)
AS
SELECT
G.Id as Id,
G.ParentId as ParentId,
G.GParentId as GParentId,
G.GGParentId  as GGParentId,
G2.ParentId as GGGParentId
FROM Hierarchy2 G left join Hierarchy2 G2
ON G.GGParentId=G2.Id;

/*
 * The output of Hierarchy3 is used as the input for Hierarchy4, the query does a
 * self-join for the fourth hierarchy
 */
CREATE OUTPUT WINDOW Hierarchy4
SCHEMA(Id string,ParentId string,GParentId string,GGParentId string,GGGParentId
string,GGGGParentId string)
PRIMARY KEY(Id)
AS
SELECT
G.Id as Id,
G.ParentId as ParentId,
G.GParentId as GParentId,
G.GGParentId  as GGParentId,
G.GGGParentId as GGGParentId,
G2.ParentId as GGGGParentId
FROM Hierarchy3 G left join Hierarchy3 G2
ON G.GGGParentId=G2.Id;

/*
 *I am interested only in rows which has all four entities parent,G-parent,
 *GG-parent,GGG-parent,GGGG-parent filter out all the rows which has null values.
 */
CREATE OUTPUT WINDOW FlattenHierarchy
SCHEMA (Id string,ParentId string,GParentId string,GGParentId string,GGGParentId
string,GGGGParentId string)
PRIMARY KEY(Id)
AS SELECT *
FROM Hierarchy4 G
WHERE isnull(G.GParentId)=0 and  isnull(G.GGParentId)=0 and isnull(G.GGGParentId)=0 and
isnull(G.GGGGParentId)=0;
```

Method 2*:*

```
CREATE INPUT WINDOW GroupEntity
SCHEMA(Id string,ParentId string,Type_t string, LossFactor float,WarningThreshold
float,LimitThreshold float)
PRIMARY KEY(Id);

/*
 * This is a single query solution for the problem stated. The 5 queries of method1
 * can be replaced with a single query.
 */
CREATE OUTPUT WINDOW FlattenHierarchy2
SCHEMA(Id string,ParentId string,GParentId string,GGParentId string,GGGParentId
string,GGGGParentId string)
PRIMARY KEY(Id)
AS
```

```
SELECT
G.Id as Id,
G.ParentId as ParentId,
G1.ParentId as GParentId,
G2.ParentId as GGParentId,
G3.ParentId as GGGParentId,
G4.ParentId as GGGGParentId
FROM GroupEntity G left join (GroupEntity G1 left join (GroupEntity G2 left join
       (GroupEntity G3 left join GroupEntity G4 ON G3.ParentId=G4.Id) ON
       G2.ParentId = G3.Id) ON G1.ParentId =G2.Id ) ON G.ParentId=G1.Id
WHERE isnull(G1.ParentId)=0 and isnull(G2.ParentId)=0 and isnull(G3.ParentId)=0 and
isnull(G4.ParentId)=0
;
```

5: HOW TO SPLIT ROWS IN A BUNDLE USING CCL

Problem: In some cases, a particular event can result in multiple rows being grouped together in a bundle or single transaction. In ESP, such bundles can occur when you have a query that joins a stream to a window. Occasionally, there is a problem with the bundle whereby a row fails to meet criteria further downstream and ESP rejects the entire bundle. In some of these situations, we don't want the entire bundle rejected; or, we want to split the bundle into separate messages for individual processing. In these cases, the solution is to split the bundle into separate rows.

Solution: Consider the following CCL code for splitting a bundle into separate rows:

```
/* Assuming data coming in one bulk or transaction through the input stream */
CREATE INPUT STREAM s1 schema (symbol string, volume integer, price float);

/*
 *Break the batching using a flex stream and a dictionary
 *this will output 1 row at a time every 1 second
 */
CREATE FLEX flex1
IN s1
OUT OUTPUT STREAM break_batch
SCHEMA(symbol string, volume integer, price float)
BEGIN
DECLARE
       dictionary(long,typeof(s1)) dict;
       typeof(s1) record;
       long id:=1;
       long fill_id:=1;
END;
ON s1
{
       dict[s1.ROWID]:=s1;
       print('inside dictionary',s1.symbol);
       fill_id++;
};
EVERY 1 second
{
       if(fill_id=id)
       {
       exit;
       }
       else
       {
       record:=dict[id];
       print('inside every 1 second',record.symbol);
       output setOpcode(record,insert);
       remove(dict,id);
       id++;
       }
};
END;
```

6: HOW TO WAIT FOR A DB APPLICATION TO FINISH A CALCULATION AND RETURN A RESULT

Problem: There are many times when we want to take advantage of the computational powers of a database. In these cases, we will need to insert data into a database table, perform a computation and then retrieve the results for further processing by ESP.

Solution: Below, is an ESP coding sample that creates a table within Hana, sends it data, requests HANA to recalculate a value (or a time based range of values, such as futures forecast) and then waits for the results to be used in subsequent processing:

```
CREATE SCHEMA NEW_schema1 ( M_ID integer, DT timestamp, Symbol string , Price money(2) )
;
CREATE SCHEMA NEW_schema2 ( M_ID integer, DT timestamp, Symbol string , Forecast integer
,  ConvPrice money(2) ) ;

CREATE INPUT WINDOW SendtoHana SCHEMA NEW_schema1
PRIMARY KEY (M_ID)
KEEP 100 Rows
;

CREATE OUTPUT WINDOW OutputToHana PRIMARY KEY DEDUCED keep 100 rows  AS SELECT * FROM
SendtoHana ;

CREATE FLEX flexOpResponsefromHana
IN OutputToHana
OUT OUTPUT WINDOW ResponsefromHana SCHEMA NEW_schema2
PRIMARY KEY (M_ID,DT)
KEEP 100 Rows
BEGIN
DECLARE
/* The typedef must be the same as the return value of the stored procedure. */
typedef [ integer M_ID;  timestamp DT;| string Symbol ; integer Forecast ; money(2)
ConvPrice;] rec;
/* v is a vector variable to store the return value of the stored procedure  */
vector(rec) v;
END;

ON OutputToHana
{
string ForecastToCheck := to_string(OutputToHana.M_ID);

[ integer M_ID;  timestamp DT;| string Symbol ; integer Forecast ; money(2) ConvPrice;]
newrec;

       /*
* initialises v using previous record sent to HANA, because newrec does not exist * yet
*/
       v := new vector(rec);
       newrec := v[0];

       /* loop whilst no vector values have been retrieved */
       while (isnull(v[0])) {
       getData(v,'HANAODBCService','SELECT * FROM "DAVID"."NEW52" WHERE "Forecast" =
?',ForecastToCheck );
       }

/* now output all records that have been retrieved */
       for(result in v){
              output (result);
              }
              /* reset the cache */
              resize (v,0);
};
END;


CREATE OUTPUT WINDOW SendToDownstreamConsumer
```

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

ATTACH OUTPUT ADAPTER HANAOut TYPE hana_out TO OutputToHana
PROPERTIES service = 'HANAODBCService' ,
table = 'NEW51'
```

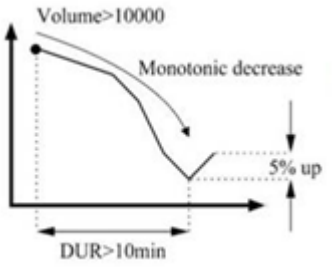
7: HOW TO SPOT A TRADING PATTERN IN CCL

Problem: Technical analysis, also known as charting, is the study of the trading history (the price and volume over time) of any type of traded market (stocks, commodities, etc.) to attempt to predict future prices. Technical analysts look for many different patterns in price movements. The interpretations of such patterns are used to support trading decisions.

Investors who are able to spot trends before other market players can make early moves and thus increase their trading profit margins. ESP can be used to perform real-time monitoring of complex patterns across many different securities.

Consider the following situation:

1. A stock ticker stream with schema (Name, Price, Volume, Tradetime).
2. A large trade for a company (volume > 10,000), followed by a monotonic decrease of the stock price of this company for at least 10 minutes. After the decrease, the stock suddenly rebounds (increases at least 5% in value).



3. Goal: Design a query that monitors the ticker stream to find the above pattern for any listed company.

Solution: Consider the following query that monitors the sticker stream to find the pattern shown above for any listed company:

```
/* TradeTime column is used for the playback control
 * rate of playback can be changed without having to change retention
 * and periods
 */
CREATE INPUT STREAM Trades
SCHEMA (Symbol string,Volume integer,Price float,TradeTime timestamp)
;

/* Adds the time_initial field to record the time at which an event arrived */
CREATE OUTPUT STREAM ticker_withtimestamp
SCHEMA(Symbol string,Volume integer,Price float,time_initial bigdatetime)
AS
SELECT
T.Symbol as Symbol,
T.Volume as Volume,
T.Price as Price,
now() as time_initial
FROM Trades T;

/*State 2 */
/* This is the second state, we need the previous Price for every symbol
```

```
 * so we can check the atomic decrease. Adds eventcache and column prev_Price
 * to store the previous Price of the same symbol, this will be used to check
 * atomic decrease later
 */

CREATE OUTPUT STREAM update_prev_Price
SCHEMA(Symbol string,Volume integer,Price float,
       prev_Price float, time_initial bigdatetime
)
DECLARE
eventcache (ticker_withtimestamp[Symbol],2 events) cache;
END
AS
SELECT
S.Symbol as Symbol,
S.Volume as Volume,
S.Price as Price,
last(cache.Price) as prev_Price,
S.time_initial as time_initial
FROM ticker_withtimestamp  S
;

/* State 1 */
/* This is the first state when the Volume is > 10000
 * The retention policy for the window should be a little more than 10 minutes
 *  so the final state can be checked
 */
 /** Test for Vol GT 10000 */
CREATE OUTPUT WINDOW ticker_with_firststate
SCHEMA (Symbol string,Volume integer,Price float,time_initial bigdatetime)
PRIMARY KEY DEDUCED
KEEP 12 minutes
AS
SELECT *
FROM ticker_withtimestamp
WHERE ticker_withtimestamp.Volume > 10000
GROUP BY ticker_withtimestamp.Symbol
 ;

/*
 * Check if first state is satisfied. If state1 is satisfied
 * (that is Volume > 10000 ) then data flows into the state2 check.
 * A simple join with the window  determines if state 1 is satisfied.
 */
/** tests for symbols whose VOl GT 10000 within the last 10minutes */
CREATE OUTPUT STREAM check_if_firststate_satisfied
SCHEMA (Symbol string,Volume integer,
        Price float,prev_Price float,
             orig_Price float, time_initial bigdatetime,
             time_now bigdatetime)
AS
SELECT
 S.Symbol as Symbol,
 S.Volume as Volume,
 S.Price as Price,
 S.prev_Price as prev_Price,
 W.Price as orig_Price,
 W.time_initial as time_initial,
 S.time_initial as time_now
FROM update_prev_Price S,ticker_with_firststate W
WHERE S.Symbol = W.Symbol
 ;

/*State 3 */
/*
 * This is the final state. Every message is checked with first
 * state window(ticker_with_firststate) check with second state
 * window (ticker_with_secondstate)
```

```
 */
 /** checks with first state window with a join, if this gets satisfied
  * then the output stream will have data
  */
CREATE OUTPUT STREAM check_if_firststate_satisfied_2
SCHEMA (Symbol string,Volume integer,Price float,orig_Price float,
       time_initial bigdatetime,time_now bigdatetime
)
AS
SELECT
 S.Symbol as Symbol,
 S.Volume as Volume,
 S.Price as Price,
 W.Price as orig_Price,
 W.time_initial as time_initial,
 S.time_initial as time_now
FROM ticker_withtimestamp S,ticker_with_firststate W
WHERE S.Symbol = W.Symbol
 ;

/* Passes events provided that prices are decreasing by checking if the
 * Price is less than the original Price (Price when Volume >10000) and
 * if the Volume is < 10000. Calculates the time difference between when
 * the Volume was > 10000 and now. This field is needed to check if the
 * Price of the symbol is decreasing for the next 10 minutes.
 */
CREATE OUTPUT STREAM stream_for_secondstate
SCHEMA (Symbol string,Volume integer,Price float,prev_Price float,
       time_difference_seconds long
)
AS
SELECT
 S.Symbol as Symbol,
 S.Volume as Volume,
 S.Price as Price,
 S.prev_Price as prev_Price,
 timeToSec(S.time_now) - timeToSec (S.time_initial) as time_difference_seconds
FROM check_if_firststate_satisfied S
WHERE S.Volume < 10000 and S.Price < S.orig_Price
;

/* This is the second state window. Output to the window if the Price is falling
 *  continuously for the next 10 minutes. The time check S.time_difference_seconds < 610
 *  and S.prev_Price > S.Price will take care of this. When the previous
 * Price is > than current Price then the atomic decrease ceases and the window
 *  content gets deleted.
 */
CREATE OUTPUT WINDOW ticker_with_secondstate
SCHEMA (Symbol string,Volume integer,Price float,time_difference_seconds long)
PRIMARY KEY DEDUCED
AS
SELECT
S.Symbol as Symbol,
S.Volume as Volume,
S.Price as Price,
S.time_difference_seconds as time_difference_seconds
FROM stream_for_secondstate S
GROUP BY S.Symbol
//S.time_difference_seconds can be reduced for testing purposes
HAVING S.time_difference_seconds < 610 and S.prev_Price > S.Price
;


/*
 *check with second state window with a join, if this gets satisfied then
 * the output stream will have data watch this stream to see the history of
 * the pattern emerge
 */
```

```
CREATE OUTPUT STREAM check_if_secondstate_satisfied_2
SCHEMA(Symbol string, Volume integer,
        max_Price float, min_Price float,Price float,
             time_difference_with_state1 long,time_difference_with_state2 long,
             time_initial bigdatetime)
AS
SELECT
 S.Symbol as Symbol,
 S.Volume as Volume,
 S.orig_Price as max_Price,
 W.Price as min_Price,
 S.Price as Price,
 timeToSec(S.time_now) - timeToSec (S.time_initial) as time_difference_with_state1,
 W.time_difference_seconds as time_difference_with_state2,
 S.time_initial as time_initial
FROM check_if_firststate_satisfied_2 S, ticker_with_secondstate W
WHERE S.Symbol = W.Symbol;


/*
 *The final result should be after 10 minutes of atomic decrease so we
 * check time difference is >600 and the Price has increased by 1.05 * min_price
 */
CREATE OUTPUT STREAM result
SCHEMA(Symbol string,max_Price float,min_Price float,final_Price float)
AS
SELECT
S.Symbol as Symbol,
S.max_Price as max_Price,
S.min_Price as min_Price,
S.Price as final_Price
FROM check_if_secondstate_satisfied_2 S
WHERE S.Price > (1.05 * S.min_Price) and S.time_difference_with_state1 > 600
;
```

© 2013 SAP AG. All rights reserved.

SAP, R/3, SAP NetWeaver, Duet, PartnerEdge, ByDesign, SAP BusinessObjects Explorer, StreamWork, SAP HANA, 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.

Business Objects and the Business Objects logo, BusinessObjects, Crystal Reports, Crystal Decisions, Web Intelligence, Xcelsius, and other Business Objects products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of Business Objects Software Ltd. Business Objects is an SAP company.

Sybase and Adaptive Server, iAnywhere, Sybase 365, SQL Anywhere, and other Sybase products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of Sybase Inc. Sybase is an SAP company.

Crossgate, m@gic EDDY, B2B 360°, and B2B 360° Services are registered trademarks of Crossgate AG in Germany and other countries. Crossgate is an SAP company.

All other product and service names mentioned are the trademarks of their respective companies. Data contained in this document serves informational purposes only. National product specifications may vary.

These materials are subject to change without notice. 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.