SYBASE®
An SAP® Company

用例ガイド

Sybase Event Stream Processor 5.0

ドキュメント ID： DC01751-01-0500-01
改訂： 2011 年 8 月

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

目次

第 1 章　用例を使用した CCL の学習

このマニュアルは、Sybase® Event Stream Processor に付属する CCL の例の参照ガイドとして作成されました。

このマニュアルでは、プロジェクト内の特定のタスクを行うために使用する一連の CCL 要素について説明しています。サンプル・コードを使用して、タスクに最も関連のあるコードの部分を強調しています。 デフォルトでは、読み取り対象のサンプル・ファイルおよびデータ・ファイルは `C:¥<installation directory>¥ESP¥examples` にあります。 このディレクトリはインストール中に設定できます。

ESP Studio で単純なプロジェクトの例を入手できますが、このマニュアルでは説明しません。 これは、[Learning] パースペクティブからロードして実行できます。

第 2 章　アダプタの例

Event Stream Processor には、さまざまな機能を示すアダプタ関連の CCL の例をいくつか備えています。これには、アダプタの付加方法やスキーマ継承の実行方法が含まれます。

注意： 構文例は、スペースの制約により折り返されることがあります。 折り返された行は、1 行に入力してください。

ATTACH ADAPTER 文

ATTACH ADAPTER 文を使用して、File CSV Input アダプタをウィンドウに付加します。

この例では、スキーマ `TradeSchema` と、このスキーマを参照する入力ウィンドウ `TradeWindow` を作成します。

この例では次に、File CSV Input アダプタを `TradeWindow` に付加します。

この **ATTACH ADAPTER** インスタンスの名前は `csvInConn1` ですが、任意の名前を割り当てることができます。 `TYPE` 条件は、アダプタに一意であるアダプタ ID を参照します。 File CSV Input アダプタの ID は、`dsv_in` です。 この例では、デフォルト値を維持するか、必要に応じてデフォルト値を変更するかのいずれかによって、アダプタ・パラメータの値を定義します。 アダプタのタイプまたは ID と各アダプタのパラメータのリストは、『アダプタ・ガイド』で確認できます。

```
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 文

ADAPTER START GROUPS 文を使用して、プロジェクト内でのアダプタの起動順序を指定します。

この例では、スキーマ TradeSchema、CompanySchema、および JoinSchema (TradeSchema からスキーマを継承) を作成します。 カッコ内のテキストは、カラム Company を追加して TradeSchema を拡張するようにプロジェクト・サーバに指示します。

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

この例では、TradeSchema を参照する入力ウィンドウ TradeWindow と、CompanySchema を参照する入力ウィンドウ CompanyInfo を作成します。 JoinSchema で定義された構造を使用する出力ジョイン・ウィンドウが作成されます。このウィンドウは、TradeWindow と CompanyInfo の入力ウィンドウの記号とタイムスタンプの値を使用して、これらをジョインします。

```
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
          ;
```

この例では、File CSV Input アダプタ csvTradesIn2 を TradeWindow に付加し、File CSV Input アダプタ csvCompanyIn を CompanyInfo に付加します。 アダプタ・インスタンス csvTradesIn2 が RunGroup0 に割り当てられ、アダプタ・インスタンス csvCompanyIn が RunGroup1 に割り当てられます。

ADAPTER START GROUPS 文は、アダプタの起動順序を指定する際に、これらのアダプタ・グループの割り当てを使用します。 この例では、プロジェクト・サーバは最初に RunGroup1 アダプタを起動して、次に RunGroup0 アダプタを起動します。

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

スキーマ継承

既存のスキーマの構造を継承するように新しいスキーマに指示します。

この例では、スキーマ TradeSchema を作成します。

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

この例では、スキーマ VTradeSchema を作成し、**INHERITS** 構文を使用して TradeSchema カラムの値を組み込むことにより VTradeSchema を拡張します。

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

この例では、File CSV Input アダプタを付加する対象となる入力ウィンドウ TradeWindow を作成します。

最後に、この例では、集約出力ウィンドウ VwapWindow を作成します。このウィンドウには、TradeWindow のデータの出来高加重平均価格が返されます。 返される値は Symbol でグループ分けされます。

```
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;
```

opcode を含むアダプタ・データ

File CSV Input アダプタに、**expectStreamNameOpcode** アダプタ・プロパティを使用します。

この例では、以下のデータ・セットを使用します。

```
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
```

データの i、d、u の値は、それぞれデータの挿入、削除、更新を示す opcode です。

この例では、File CSV Input アダプタを付加する対象となるデータの入力ウィンドウ `win1` を作成します。

実行する必要がある opcode が受信データにあることをプロジェクト・サーバが認識するように、アダプタ・プロパティ `expectStreamNameOpcode` は true に設定されています。

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

File CSV Output アダプタ

File CSV Output アダプタを使用して、外部の送信先にデータを送信します。

この例では、入力ウィンドウ `InTrades` によって参照されるスキーマ `TradeSchema` を作成します。 この例では、File CSV Output アダプタ `csvOut` と File CSV Input アダプタ `InConn` を `InTrades` に付加します。

```
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 アダプタ

Database Input アダプタを使用してデータベースに接続します。

前提条件
この例を実行するには、サポートされている構文を使用してデータベースに `Trades` テーブルを作成します。 このテーブルには以下の値を指定してください。

カラム	データ型	値
Ts	datetime	not null
Symbol	char(4)	not null
Price	money	not null
Volume	int	not null

また、`Trades` (`Ts`) にユニーク・インデックス `ind1` を作成し、`Trades` のすべてのパーミッションを public に付与します。

最後に、以下の例をモデルにして <ESP_HOME>/bin の `services.xml` ファイルを設定します。

```
<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="ConnectString"></Parameter>
                <Parameter Name="ConnectionPoolSize">-1</Parameter>
        </Service>
```

この例を実行する前にテーブルにデータを移植します。

例
この例では、スキーマ `TradeSchema` を作成してから、それぞれ `TradeSchema` を参照する入力ウィンドウ `TradeWindow` と出力ウィンドウ `TradeOutWindow` を作成します。**`SELECT`** `*` (すべてを選択) 構文は、`TradeWindow` で処理されるすべてのデータを `TradeOutWindow` に出力するようにプロジェクト・サーバに指示します。

この例では、Database Input アダプタを TradeWindow に付加し、前提条件として設定されたデータベースからデータを読み取ります。

```
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 アダプタ

Database Output アダプタを使用して、外部のデータベースにデータを送信します。

前提条件
この例を実行するには、サポートされている構文を使用してデータベースに VwapWindow テーブルを作成します。 このテーブルには以下の値を指定してください。

カラム	データ型	値
Symbol	char(4)	not null
Price	money	not null

また、Trades (Ts) にユニーク・インデックス ind1 を作成し、VwapWindow のすべてのパーミッションを public に付与します。

最後に、以下の例を設定のモデルにして <ESP_HOME>/bin の services.xml ファイルを設定します。

```
<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="ConnectString"></Parameter>
                <Parameter Name="ConnectionPoolSize">-1</Parameter>
        </Service>
```

テーブルには、File CSV Input アダプタからのデータが自動的に移植されます。

例
この例では、スキーマ TradeSchema を作成してから、TradeSchema を参照する入力ウィンドウ TradeWindow を作成します。

この例では、集約出力ウィンドウ VwapWindow を作成します。このウィンドウには、TradeWindow のデータの出来高加重平均価格が返されます。 返される値は Symbol でグループ分けされます。

```
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;
```

この例では、Database Output アダプタを VwapWindow に付加します。 プロジェクト・サーバは、日付フォーマットで日付の値を処理します。つまり、日付の値がトランケートされます。

```
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 ;
```

この例では、File CSV Input アダプタを TradeWindow に付加し、外部ソースからデータを読み取り、前提条件として設定されたデータベースに移植します。

```
 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';
```

ポーリング機能を持つ Database Input アダプタ

Database Input アダプタを使用してデータベースに接続し、データベースをポーリングします。

前提条件

この例を実行するには、サポートされている構文を使用してデータベースに `Trades` テーブルを作成します。 このテーブルには以下の値を指定してください。

カラム	データ型	値
Ts	datetime	not null
Symbol	char(4)	not null
Price	money	not null
Volume	int	not null

また、`Trades` (`Ts`) にユニークなノンクラスタード・インデックス `ind1` を作成し、`Trades` のすべてのパーミッションを public に付与する必要があります。

最後に、以下の例を設定のモデルにして `<ESP_HOME>/bin` の `services.xml` ファイルを設定します。

```
<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="ConnectString"></Parameter>
                  <Parameter Name="ConnectionPoolSize">-1</Parameter>
        </Service>
```

テーブルにデータを移植して、この例を実行します。

例

この例では、スキーマ `TradeSchema` を作成してから、それぞれ `TradeSchema` を参照する入力ウィンドウ `TradeWindow` と出力ウィンドウ `TradeOutWindow`

を作成します。**SELECT** * (すべてを選択) 構文は、TradeWindow で処理されるすべてのデータを TradeOutWindow に出力します。

この例では、Database Input アダプタを TradeWindow に付加し、前提条件として設定されたデータベースからデータを読み取ります。 このアダプタ・インスタンスのポーリング間隔が 10 である場合は、10 秒ごとにデータベースに新しい内容がポーリングされます。

```
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' ;
```

第 3 章　ストリームとウィンドウの例

Event Stream Processor には、さまざまな機能を示すストリームとウィンドウの例をいくつか備えています。これには、デルタ・ストリームの使用方法、ジョインと union の作成方法、ストリームの分割方法が含まれます。

注意： 構文例は、スペースの制約により折り返されることがあります。 折り返された行は、1 行に入力してください。

ストリーム

入力ストリームとローカル・ストリームを作成します。

この例では、入力ストリーム `TradeStream` とローカル・ストリーム `TradeLocalStream` を作成します。 ローカル・ストリームは、**SELECT** * (すべてを選択) 構文を使用して `TradeStream` からすべてのデータ・カラムを取得します。

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

AS
SELECT * from TradeStream;
```

この例では、File CSV Input アダプタを `TradeStream` に付加してから、出力ストリーム `TradeOutStream` を作成します。

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

`TradeOutStream` は、**SELECT** * (すべてを選択) 構文を使用して `TradeLocalStream` からすべてのデータ・カラムを取得します。また、File CSV Output アダプタを使用して、これらのデータ・カラムを出力します。

```
ATTACH OUTPUT ADAPTER Adapter1
   TYPE dsv_out
   TO TradeOutStream
   PROPERTIES
     dir = '$ProjectFolder/../output' , file = 'streams.csv' ,
outputBase = TRUE ,     hasHeader = TRUE , runtimeDir = 'c:/esp/
output' ;
```

ローカル・ウィンドウと出力ウィンドウ

ストリームとウィンドウを比較して、ローカル・ウィンドウと出力ウィンドウの差異を確認します。

この例では、スキーマ `TradeSchema` を作成してから、`TradeSchema` を参照する入力ウィンドウ `TradeWindow` を作成します。 `TradeWindow` に File CSV Input アダプタが付加されます。

この例では次に、一連のローカル・ストリームと出力ストリームおよびローカル・ウィンドウと出力ウィンドウを作成します。 出力ストリームと出力ウィンドウはパブリックです。これらはアダプタを使用して外部データ・ソースと通信します。 ローカル・ストリームとローカル・ウィンドウは内部でのみ表示されます。これらにアダプタを付加することはできません。

```
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 ;
```

デルタ・ストリーム

デルタ・ストリームは、**getrowid** と **now** の関数が組み込まれています。

この例では、File CSV Input アダプタを付加する対象となる入力ウィンドウ `TradesWindow` を作成します。

この例では次に、デルタ・ストリーム `DeltaTrades` を作成します。また、**SELECT** 句を使用して、**getrowid** と **now** の関数を `TradesWindow` に適用します。

getrowid 関数は、入力ウィンドウの株式記号、タイムスタンプ、価格、値のローのシーケンス番号を取得します。 **now** 関数は、`bidgatetime` フォーマットで処理日付をパブリッシュします。

```
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
```

この例では、結果を表示する出力ウィンドウ `TradesOut` を作成します。

ウィンドウのジョイン

FROM 句と ANSI **JOIN** 構文を使用して 2 つのウィンドウをジョインします。

この例では、2 つのスキーマ `StocksSchema` と `OptionsSchema`、および出力ウィンドウ `OutSchema` を作成します。

この例では次に、2 つの出力ウィンドウ `InStocks` と `InOptions` を作成します。これらのウィンドウはそれぞれ `StocksSchema` で定義された構造と `OptionsSchema` で定義された構造を使用します。

最後に、この例では、`OutSchema` で定義された構造を使用して出力ジョイン・ウィンドウを作成します。このウィンドウは、`InStocks` と `InOptions` の入力ウィンドウの記号とタイムスタンプの値を使用して、これらをジョインします。

```


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 ;
```

ストリームのジョイン

2 つのウィンドウを 1 つのストリームにジョインします。

この例では、2 つのスキーマ StocksSchema と OptionsSchema を作成してから、StocksSchema を参照する入力ウィンドウ InStocks と、OptionsSchema を参照する入力ウィンドウ InOptions を作成します。

この例では、出力ジョイン・ストリーム OutStockOption を作成します。このストリームは、InStocks と InOptions の入力ウィンドウの記号の値を使用して、これらをジョインします。

```
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
;
```

この例では、2 つの **ATTACH ADAPTER** インスタンス csvInConn1 と csvInOptions を作成します。 一方のインスタンスの InStocks ウィンドウと、もう一方のインスタンスの InOptions ウィンドウに File CSV Input アダプタが付加されます。

最後に、File CSV Output アダプタ Adapter1 を OutStockOptions に付加し、ストリームのジョインの結果をパブリッシュします。

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

外部ジョイン

入力ウィンドウ間で、左ジョイン、右ジョイン、フル・ジョインを作成します。

この例では、2 つのスキーマ StocksSchema と OptionsSchema を作成します。次に、StocksSchema を参照する入力ウィンドウ InStocks と、OptionsSchema を参照する入力ウィンドウ InOptions を作成します。

この例では、出力ウィンドウ OutStockOptionFOJ を作成します。このウィンドウは、InStocks と InOptions のタイムスタンプ値を使用して、これらウィンドウ間のフル・ジョインを作成します。

```
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;
```

この例では、出力ウィンドウ OutStockOptionLOJ を作成します。このウィンドウは、InStocks と InOptions のタイムスタンプ値を使用して、これらウィンドウ間の左外部ジョインを作成します。

```
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 ;
```

この例では、出力ウィンドウ OutStockOptionROJ を作成します。このウィンドウは、InStocks と InOptions のタイムスタンプ値を使用して、これらウィンドウ間の右外部ジョインを作成します。

```
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 ;
```

この例では、File CSV Input アダプタ csvInStocks を InStocks に付加し、File CSV Input アダプタ csvInOptions を InOptions に付加します。

ストリームの union

2 つのウィンドウ間に単純な union を作成します。

この例では、2 つのスキーマ StocksSchema と OptionsSchema を作成します。これらのスキーマはそれぞれ入力ウィンドウ InStocks と InOptions の構造を定義します。

この例では次に、出力ウィンドウ Union1 を作成します。このウィンドウは、InStocks と InOptions の入力ウィンドウ間の union を作成します。

```
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
;
```

この例は、2 つの **ATTACH ADAPTER** インスタンス `csvInConn1` と `csvInConn2` を作成することによって完了します。 一方のインスタンスの `InStocks` ウィンドウと、もう一方のインスタンスの `InOptions` ウィンドウに File CSV Input アダプタが付加されます。

ストリーム分割

複数の出力ウィンドウを使用してストリーム分割を実行します。

ストリーム分割を使用すると、1 つのストリームから複数のストリームにデータを送信できます。

この例では、スキーマ `TradeSchema` を作成し、そのスキーマを入力ウィンドウ `TradeWindow` に適用します。

この例では次に、3 つの出力ウィンドウ `OutMyTrades`、`OutBigTrades`、`OutOtherTrades` を作成します。これらのウィンドウは、ウィンドウ間で `TradeWindow` のデータを分割します。

```
CREATE  OUTPUT  WINDOW OutMyTrades
    SCHEMA  TradeSchema
    PRIMARY KEY (Ts)
AS
    SELECT * from TradeWindow
    WHERE TradeWindow.Symbol IN ('IBM', 'EBAY') ;
```

`OutMyTrades` は、`TradeWindow` から記号 `IBM` または `EBAY` を含むデータを出力します。

```
CREATE  OUTPUT  WINDOW OutBigTrades
    SCHEMA  TradeSchema
    PRIMARY KEY (Ts)
AS
    SELECT * from TradeWindow
    WHERE TradeWindow.Price * TradeWindow.Volume > 100000 ;
```

`OutBigTrades` は、`TradeWindow` から `TradeWindow.Price * TradeWindow.Volume` の積が 100,000 より大きいデータを出力します。

```
CREATE  OUTPUT  WINDOW OutOtherTrades
    SCHEMA  TradeSchema
    PRIMARY KEY (Ts)
AS
SELECT * from TradeWindow
 WHERE  NOT (( TradeWindow.Price * TradeWindow.Volume > 100000 )
    OR (TradeWindow.Symbol IN ('IBM', 'EBAY') )
  )
 ;
```

`OutOtherTrades` は、前の 2 つの出力ウィンドウで指定された条件を満たさないデータ・セットをすべて出力します。

この例は、受信ストリーム・データを処理するために File CSV Input アダプタを `TradeWindow` に付加することによって完了します。

第 4 章　関数の例

Event Stream Processor には、さまざまな機能を示す関数の例を備えています。これには、ビット処理関数や基本集合関数の使用方法が含まれます。

注意： 構文例は、スペースの制約により折り返されることがあります。 折り返された行は、1 行に入力してください。

CREATE LIBRARY 文

外部ライブラリを識別し、そのライブラリに関数を配置します。

この例では、Event Stream Processor に付属のライブラリ・ファイル `Functions.class` を使用します。 外部ファイルを使用して ESP Studio 内にライブラリを作成する場合は、CLASSPATH 変数にライブラリ・ファイルのソース・ディレクトリを含めてください。 ESP Studio を使用しない場合は、プロジェクト設定ファイル (`.ccr`) を編集して、Java-classpath オプションをライブラリ・ファイルのソース・ディレクトリに設定できます。

この例は、`Functions.class` ファイルから Java 言語のライブラリ `SC1` を作成する **CREATE LIBRARY** 文で始まります。

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

この例では、2 つのスキーマ `Schema1` と `OutSchema` を作成します。 この例では次に、`Schema1` を参照する入力ウィンドウ `win1` と、`OutSchema` を参照する出力ウィンドウ `OutWin` を作成します。 データを手動で `win1` にロードします。

```
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
;
```

集合関数

送信データに **first**、**last**、**max**、**min** の各関数を適用します。

この例では、2 つのスキーマ `TradeSchema` と `OpenCloseMinMaxSchema`、および File CSV Input アダプタを付加する対象となる入力ウィンドウ `TradeWindow` を作成します。

この例では次に、`OpenCloseMinMaxSchema` で定義された構造を使用する出力ウィンドウ `OutOpenCloseMinMax` を作成します。 **SELECT** 句は、`TradeWindow` のデータから最初の値、最後の値、最小値、最大値を返します。また、`Symbol` でその結果をグループ分けします。

```
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;
```

ビット処理関数

出力ウィンドウに **bitand**、**bitor**、**bitshiftleft**、**bitshiftright**、**bitmask** の各演算を適用します。

この例では、2 つのスキーマ `IntNumbersSchema` と `ResultNumbersSchema` を作成します。

この例では、`ResultNumbersSchema` にビット処理関数を適用します。 ビット処理関数を使用すると、データを構成する個々のビットにアクセスして操作することができます。

```
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='$ProjectFolder/../data',
        file = 'Numbers1000.csv' ,
        delimiter = '' ;
```

データ集約

カンマ区切り値 (`.csv`) ファイルからデータを読み取り、出来高加重平均価格 (**vwap**) 関数を使用してデータを集約します。

この例では、入力ウィンドウ `TradeWindow` によって参照されるスキーマ `TradeSchema` を作成します。 この例では、File CSV Input アダプタを `TradeWindow` に付加します。

この例では、出力ウィンドウ VwapWindow を作成します。このウィンドウには、TradeWindow によって処理される取引値の出来高加重平均価格の結果が出力されます。 結果は Symbol でグループ分けされます。

```
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;
```

第 5 章　ストアの例

Event Stream Processor には、デフォルト・ストア、メモリ・ストア、ログ・ストアの作成方法を示す CCL の例を備えています。

注意： 構文例は、スペースの制約により折り返されることがあります。 折り返された行は、1 行に入力してください。

ストア

デフォルト・ストア、メモリ・ストア、ログ・ストアを作成します。

この例では、メモリ・ストア `MemStore`、デフォルト・ストア `DefaultStore`、ログ・ストア `LogStore` を作成します。 各ストアは、それぞれデフォルトのパラメータ値を保持します。

```
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 ;
```

この例では、`MemStore` を参照する入力ウィンドウ `TradesWindowMem` と、`SELECT *` (すべてを選択) 構文を使用して `TradesWindowMem` からすべてのデータ・カラムを取得する出力ウィンドウ `DefaultStoreWindow` を作成します。

この例では、`LogStore` を参照する出力ウィンドウ `LogStoreWindow` を作成します。`LogStoreWindow` は **SELECT** 句と **FROM** 句を使用して `TradesWindowMem` からタイムスタンプ、価格、記号、個数のデータを引き出します。

この例では、File CSV Input アダプタ `InConn` を `TradesWindowMem` に付加します。

前払い請求書作成機

携帯電話プラン用のサンプルの前払い請求書作成アプリケーションを構築します。

この例では、一連のメモリ・ストア StaticStore、CDRsStore、AccountCDRsStore、AccountSummariesStore、AuthsStore、AccountAuthStore、AccountAuthsMinsStore を作成します。

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

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

この例では、2 つの入力ウィンドウ Accounts と CallPlans、および出力ウィンドウ AccountPlans を作成します。これらのウィンドウはすべて StaticStore を参照します。AccountPlans は、Accounts と CallPlans のコール・プランとプラン・タイプの値を使用して、これらウィンドウ間のジョインを作成します。

```
CREATE  OUTPUT  WINDOW AccountPlans
SCHEMA (AccountId INTEGER, MonthlyRate FLOAT,
    PlanMinutes FLOAT, AddlMinutesRate FLOAT, PrepaidTotal FLOAT)
PRIMARY KEY (AccountId)
 STORE StaticStore
 AS
SELECT  Accounts.AccountID  AS AccountId,  CallPlans.MonthlyRate  AS
MonthlyRate,
    CallPlans.PlanMinutes  AS PlanMinutes,
CallPlans.AddlMinutesRate  AS AddlMinutesRate,
    Accounts.PrepaidTotal  AS PrepaidTotal
  FROM Accounts JOIN CallPlans
 ON Accounts.CallPlan = CallPlans.CallPlanType;
```

この例では、CDRsStore を参照する入力ウィンドウ CDRs と、AccountCDRsStore を参照する出力ウィンドウ AccountSummariesJoin を作成します。CDRs はコール・データ・レコードを参照します。AccountSummariesJoin は CDRs と AccountPlans の請求書タイプ・コード (BillTypCd) とアカウント ID の値を使用して、これらウィンドウ間のジョインを作成します。

この例では、AccountSummariesStore を要約する出力ウィンドウ AccountSummaries を作成します。AccountSummaries は **SELECT** 句と **FROM** 句を使用して AccountSummariesJoin からデータを引き出し、アカウント・プラン ID でデータをグループ分けします。

```
CREATE  OUTPUT  WINDOW AccountSummaries
SCHEMA (AccountId INTEGER, MonthlyRate FLOAT, TotalRatedUsage FLOAT,
```

```
TotalMinutes FLOAT, CallCount INTEGER)
 PRIMARY KEY DEDUCED
 STORE AccountSummariesStore
 AS
SELECT AccountSummariesJoin.AccountPlansAccountId AS AccountId,
    AccountSummariesJoin.AccountPlansMonthlyRate AS MonthlyRate,
    (( ( (sum(AccountSummariesJoin.CDRsCallDuration) >
AccountSummariesJoin.AccountPlansPlanMinutes) )
*AccountSummariesJoin.AccountPlansAddlMinutesRate) *
(sum(AccountSummariesJoin.CDRsCallDuration) -
AccountSummariesJoin.AccountPlansPlanMinutes)) AS TotalRatedUsage,
    sum(AccountSummariesJoin.CDRsCallDuration) AS TotalMinutes,
    count(AccountSummariesJoin.CDRsCallDuration) AS CallCount
FROM AccountSummariesJoin
GROUP BY AccountSummariesJoin.AccountPlansAccountId;
```

この例では、AccountAuthsStore を参照する出力ウィンドウ AccountAuthsMinsJoin を作成します。AccountAuthsMinsJoin は、AccountPlans と AccountSummaries の請求書タイプとアカウント ID の値を使用して、これらウィンドウ間のジョインを作成します。

この例では、AccountAuthsMinsStore を参照する出力ウィンドウ AccountAuthsMins を作成します。AccountAuthsMins は **SELECT** 句と **FROM** 句を使用して AccountAuthsMinsJoin からデータを引き出し、アカウント・プラン ID でデータをグループ分けします。

この例は、File XML Input アダプタを Accounts、CallPlans、CDRs、Auths に付加することによって完了します。

第 6 章 フレックスの例

Event Stream Processor には、さまざまな機能を示すフレックスの例をいくつか備えています。これには、SPLASH 構文、opcode、タイマ、**if/then/else** 条件、イベント・キャッシュの使用方法が含まれます。

注意： 構文例は、スペースの制約により折り返されることがあります。 折り返された行は、1 行に入力してください。

フレックス・ストリームを使用したデータ管理

フレックス・ストリームを使用してデータを管理します。

この例では、3 つのスキーマ `TradeSchema`、`Totalschema`、`Tutelage` と 1 つの入力ウィンドウ `TradeWindow` を作成します。 `TradeWindow` に File CSV Input アダプタが付加されます。

この例では次に、`TradeWindow` から `OldTradeEvents` にデータを出力するフレックス・ストリーム `TrackOldTrades` を作成します。 **switch** 文は、挿入と更新の出力のみをサポートします。そのため、削除は出力ウィンドウに渡されません。

```
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; ];
              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
  ;
```

複数の入力

複数の入力に複数のフレックス・ストリームを使用します。

この例では、2 つの入力ウィンドウ `Trades 1` と `Trades 2` を作成します。

この例では次に、2 つの入力ウィンドウをジョインするフレックス・ストリーム `TradesMSFTFlexStream` を作成し、出力ウィンドウ `TradesMSFTFlexStream` を追加します。

```
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;
```

この例では、`Trades1` と `Trades2` のウィンドウをジョインするもう 1 つのフレックス・ストリーム `TradesCSCOFlexStream` を作成します。

```
 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);
 };
```

最後に、この例では、TradesMSFTFlexStream と TradesCSCOFlexStream をジョインするフレックス・ストリーム TradesPickedFlexStream を作成します。

```
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;
```

タイマを使用した平均取引価格

タイマを使用して、5 秒ごとに出力ウィンドウに新しいローを送信します。

この例では、スキーマ TradesSchema と入力ウィンドウ TradeWindow を作成します。 そのウィンドウに File CSV Input アダプタが付加されます。

この例では、TradeWindow に 10 個のローのデータ保存ポリシーを設定するフレックス・ストリーム FlexTimer を作成します。 **ON** 句は、5 秒ごとに計算

vvalue ++ を取引価格に適用するようにプロジェクト・サーバに指示します。この式は、ローカル変数 vvalue の現在の値を増加させます。

```
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;
```

DECLARE ブロックの変数

変数を定義し、通常のストリームとフレックス・ストリームの両方でその変数を使用します。

この例では、変数 ThresholdValue のデフォルト値に 1000 を指定します。

```
declare
 INTEGER ThresholdValue := 1000;
end;
```

この例では、2 つのスキーマ TradeSchema と ControlSchema を作成します。入力ウィンドウ TradeWindow は TradeSchema を参照し、入力ストリーム ControlMsg は ControlSchema を参照します。

この例では次に、出力ウィンドウ OutTradeWindow を作成します。 **SELECT** 句は、ThresholdValue より大きいローを OutTradeWindow に送信します。

```
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;
```

この例では、制御メッセージを処理するフレックス・ストリーム FlexControlStream を作成します。 **BEGIN** 構文により、制御メッセージに基づいた条件が指定されます。 制御メッセージが set である場合、

ThresholdValue の値はデフォルト値の 1000 ではなく、制御メッセージの値と等しくなるように設定されます。

```
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
;
```

最後に、この例では、File CSV Input アダプタを使用して、2 つの **ATTACH ADAPTER** インスタンス csvInCntMsg と csvInConn1 を作成します。 最初のインスタンスでは、アダプタが ControlMsg に付加され、RunGroup1 に割り当てられます。 2 番目のインスタンスでは、アダプタが TradeWindow に付加され、RunGroup2 に割り当てられます。 **ADAPTER START GROUPS** 文は、最初に制御メッセージを読み取ってから株取引のデータを読み取るように、プロジェクト・サーバに指示します。

イベント・キャッシュ

出力ウィンドウでイベント・キャッシュを使用します。

この例では、入力ウィンドウ Trades と出力ウィンドウ Last5MinuteStats を作成します。

この例では、**DECLARE** ブロックを使用して、Trades ウィンドウにイベント・キャッシュを配置します。 その結果、Last5MinuteStats ウィンドウは、記号がキャッシュされるたびに最後の 300 秒間のデータを保持します。

```
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;
```

この例では、出力ウィンドウ `Last10TradesStats` を作成します。また、**DECLARE** ブロックを使用して、`Trades` ウィンドウに別のイベント・キャッシュを配置します。 その結果、`Last10TradesStats` ウィンドウは、`Trades` ウィンドウで記号がキャッシュされるたびに最後の 10 個の取引を保持します。

```
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;
```

if/then/else を含む SPLASH

SPLASH **if/then/else** 文を使用します。また、**switch** 文を使用して同じロジックを実行します。

この例では、スキーマ `TradeSchema` と、そのスキーマを参照する入力ウィンドウ `TradeWindow` を作成します。 そのウィンドウに File CSV Input アダプタが付加されます。

この例では次に、ネストされた **if** 文で SPLASH **if/then/else** 関数を実行します。

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

これらの **if** 文は、`TradeWindow.Price * TradeWindow.Volume` の積が 1,000,000 より小さい場合に、取引のデータ値を出力するようにプロジェクト・サーバに指示します。 **else if** 文は条件が true でない場合に実行されます。

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

else if 文は、ウィンドウの株価の合計が 10,000 より小さい場合に、10 より大きい取引のデータ値を出力するようにプロジェクト・サーバに指示します。 追加の **else** 文はこれらの条件が true でない場合に実行されます。

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

else 文は、ウィンドウの株価の合計が 1,000 より小さく、上記の **if/else** 条件が true でない場合に、その出力を完了するようにプロジェクト・サーバに指示します。

この例では次に、**switch** 構文を使用してそのすべてが含まれる同じ条件を達成します。

```
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
;
```

また、**switch** 構文は、`TradeWindow.Price` の値を `float` に変換し、対数を値に適用して、それを `integer` に変換します。

getOpcode を含む SPLASH

フレックス・ストリームを使用して、項目が削除されるか、期限切れになる際にその項目を取得します。

この例では、スキーマ `TradeSchema` を作成してから、`TradeSchema` の構造を継承するスキーマ `DeleteOrExpireSchema` を作成します。 この例では、File CSV Input アダプタを付加する対象となる入力ウィンドウ `TradeWindow` を作成します。

この例では次に、`TradeWindow` から `OldTradeEvents` にデータを出力するフレックス・ストリーム `TrackOldTrades` を作成します。

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

getOpcode 関数により、ウィンドウで実行されるオペレーションが決定します。 **switch** 文は、削除の処理のみ行います。

```
 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
;
```

第 7 章 DECLARE ブロックの例

Event Stream Processor には、パラメータと関数の宣言を含む、**DECLARE** ブロックの使用方法についての例を備えています。

注意： 構文例は、スペースの制約により折り返されることがあります。 折り返された行は、1 行に入力してください。

CCL 関数

DECLARE ブロックを使用して関数を定義します。

この例では、スキーマ `TradeSchema` を作成してから、**DECLARE** ブロックを使用して変数 `Value1` と `Value2` を含む関数 `MyWeightedAverage` を宣言します。 また、この例では、ローカル変数 `Weight1` も作成します。 一連の **if** と **else if** の条件により、指定した値よりも `Value 2` が大きいか小さいかに基づいて `Weight1` の値が決定します。 結果として得られる `Weight1` の値は、**to_money** 関数内のパラメータとなります。

```
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;
```

この例では、`TradeSchema` を参照する入力ウィンドウ `TradeWindow` と、インライン・スキーマを指定する出力ウィンドウ `OutWeightedAverage` を作成します。`OutWeightedAverage` は、**avg()** 関数内で `MyWeightedAverage` 関数を使用します。

```
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
    ;
```

この例は、File CSV Input アダプタ `csvInConn1` を `TradeWindow` に付加することによって完了します。

パラメータ宣言

パラメータを宣言し、出力ウィンドウで参照します。

この例では、DECLARE ブロックでパラメータ `ThresholdValue` を宣言し、デフォルト値 1000 を設定します。ランタイムまたはプロジェクト設定ファイルでデフォルト値を変更できます。

```
DECLARE
 PARAMETER INTEGER ThresholdValue := 1000;
end;
```

この例では、入力ウィンドウ `TradeWindow` と出力ウィンドウ `TradeOutWindow` を作成します。`TradeOutWindow` は、**SELECT** 文を使用して `TradeOptMatch` からデータを引き出します。**WHERE** 句は、 `TradeWindow.Volume` の積が `ThresholdValue` パラメータの値セットより大きい場合にのみ `TradeWindow` からデータを出力するように `TradeOutWindow` に指示します。

```
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;
```

この例では、File CSV Input アダプタ `csvConn1` を `TradeWindow` に付加します。

第 8 章　データ選択の例

Event Stream Processor には、さまざまな機能を示すデータ選択の例をいくつか備えています。これには、データへの **GROUP BY** 句、**AGING** 句、**WHERE** 句の適用方法が含まれます。

注意： 構文例は、スペースの制約により折り返されることがあります。 折り返された行は、1 行に入力してください。

AGING カラム

AGING 句を使用して出力ウィンドウの経過期間カラムを設定します。

この例では、メモリ・ストア `memory1` を作成してから、`memory1` ストアを使用する入力ウィンドウ `TradesWindow` を作成します。 この例では、File CSV アダプタを `TradesWindow` に付加します。

```
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;
```

この例では、出力ウィンドウ `AgingWindow` を作成します。 出力ウィンドウの経過期間カラムは、経過期間カラムの値が 20 になるまで 10 秒ごとに値を増加させます。

```
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
;
```

時間オプションを含む AGING カラム

AGING 句を使用して入力ウィンドウに時間オプションを含む経過期間カラムを設定します。

この例では、スキーマ TradeSchema と、TradeSchema の構造を継承するスキーマ TradeAgeSchema を作成します。また、TradeAgeSchema は、3 つのカラム AgeColumn、AgeStartTime、ctime を定義します。

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

この例では、TradeSchema を参照する入力ウィンドウ TradeWindow と、TradeAgeSchema を参照する出力ウィンドウ AgeWindow を作成します。この例では、**AGES EVERY** 構文を使用して、経過期間カラムの値が 10 になるまで 6 秒ごとに AgeWindow の値を増加させます。**SELECT** 句によって AgeWindow の開始時間の条件が指定されるため、 **AGING** 句で指定された更新は、現在時刻から 6 分経過するまでは開始されません。

```
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 ;
```

この例では次に、File CSV Input アダプタ csvInConn1 を TradeWindow に付加します。

データ集約

カンマ区切り値 (.csv) ファイルからデータを読み取り、出来高加重平均価格 (**vwap**) 関数を使用してデータを集約します。

この例では、入力ウィンドウ TradeWindow によって参照されるスキーマ TradeSchema を作成します。 この例では、File CSV Input アダプタを TradeWindow に付加します。

この例では、出力ウィンドウ VwapWindow を作成します。このウィンドウには、TradeWindow によって処理される取引値の出来高加重平均価格の結果が出力されます。 結果は Symbol でグループ分けされます。

```
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;
```

フィルタを使用したデータ集約

HAVING 句を使用してウィンドウにフィルタを適用します。

この例では、File CSV Input アダプタ csvInConn1 を付加する対象となる入力ウィンドウ TradeWindow を作成します。

この例では、TradeWindow によって処理される取引値の出来高加重平均価格の結果を出力する出力ウィンドウ VwapWindow を作成します。 結果は Symbol でグループ分けされます。 **HAVING** 句は、TradeWindow にフィルタ条件を適用し、Symbol のすべての Volume の値の合計が 100,000 を超える場合にのみ **vwap** の結果をパブリッシュするようにプロジェクト・サーバに指示します。

```
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;
```

last() 関数を含む GROUP BY 句

last 関数を **SELECT** 句の結果に使用します。 **HAVING** 句の **SELECT** 句の結果を参照してください。

この例では、スキーマ TradeSchema を作成します。

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

この例では、スキーマ TradesWidthDelaySchema を作成し、**INHERITS** 構文を使用して TradeSchema の構造をローの遅延が発生した TradesWidthDelaySchema に適用します。

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

この例では、File CSV Input アダプタを付加する対象となる入力ウィンドウ TradeWindow を作成します。

この例では次に、TradesWidthDelaySchema で定義された構造を使用する出力ウィンドウ TradesWithDelay を作成します。 **SELECT** 句は、タイムスタンプ、記号、価格、個数のデータのローにローの遅延を発生させます。 **HAVING** 句は、クエリの結果にある RowDelay カラムを、ウィンドウ名を指定しないで参照します。 **HAVING** 句は、出力ウィンドウを、遅延が 10 ミリ秒を超えるローに制限します。

```

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
;
```

この例では、TradeSchema で定義された構造を使用する出力ウィンドウ OutTrades を作成します。 **GROUP BY** 文は、取引価格が前回処理した取引価格を超える場合に、選択したローを Symbol を使用して処理します。 プロジェクト・

サーバは、以前の引数に基づいて、取引価格が上昇し、トレード間の時間が 10 ミリ秒を超えたことを認識します。

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

KEEP 句

出力ウィンドウに **KEEP** 句を指定します。

この例では、入力ウィンドウ `TradesWindow` と出力ウィンドウ `KeepCountWindow` を作成します。`KeepCountWindow` は、ウィンドウ内で同時に 10 個のローを保持する **KEEP** 句を含みます。

```
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
;
```

この例では、File CSV Input アダプタ `InConn` を `TradesWindow` に付加し、File CSV Output アダプタ `OutConn` を `KeepCountWindow` に付加します。

KEEP 句と AGING 句の併用

出力ウィンドウに **KEEP** 句と **AGING** 句を指定します。

この例では、スキーマ `TradeSchema` と、`TradeSchema` の構造を継承するスキーマ `TradeAgeSchema` を作成します。また、`TradeAgeSchema` は、2 つのカラム `AgeColumn` と `AgeStartTime` を定義します。

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

この例では、`TradeSchema` を参照する入力ウィンドウ `TradeWindow` を作成します。この入力ウィンドウに File CSV Input アダプタを付加します。

最後に、この例では、TradeAgeSchema を参照する出力ウィンドウ KeepAgeWindow を作成します。KeepAgeWindow は、ウィンドウ内で同時に 20 個のローを保持する **KEEP** 句を含みます。 また、この例では、**AGES EVERY** 構文を使用して、経過期間カラムの値が 10 になるまで 3 秒ごとに KeepAgeWindow を更新します。**SELECT** 句にって AgeWindow の開始時間の条件が指定されるため、**AGING** 句で指定された更新は、現在時刻から 6 分経過するまでは開始されません。

```
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 句

出力ウィンドウで **KEEP ALL** 句を使用します。

この例では、スキーマ TradeSchema を作成します。 この例では、TradeSchema を参照する入力ウィンドウ TradeWindow を作成します。この入力ウィンドウに File CSV Input アダプタを付加します。

この例では、出力ウィンドウ KeepAllWindow を作成します。この出力ウィンドウは、**KEEP ALL** 句を使用して TradeWindow からのすべてのデータを保持し、Symbol でその結果をグループ分けします。

```
CREATE OUTPUT WINDOW KeepAllWindow
        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 句

入力ウィンドウに **KEEP LAST** 句を指定します。

この例では、入力ウィンドウ TradeWindow によって参照されるスキーマ TradeSchema を作成します。

この例では次に、TradeWindow からのデータを出力する出力ウィンドウ KeepLastWindow を作成します。KeepLastWindow は、KeepLastWindow で処理された TradeWindow の最後のローのみを保持する **KEEP** 句を含みます。

```
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
;
```

この例は、File CSV Input アダプタ csvInConn1 を TradeWindow に付加することによって完了します。

WHERE 句を使用したフィルタ

出力ウィンドウのフィルタとして、**WHERE** 句を使用します。

この例では、入力ウィンドウ TradeWindow と出力ウィンドウ TradeOutWindow を作成します。

SELECT 句は、TradeWindow からすべての (*) データ・ローを返します。 **WHERE** 句は、株式数が 10,000 より少ない場合にデータに対しフィルタを適用します。その結果、プロジェクト・サーバは、TradeWindow に 10,000 を超える株式がある場合にすべてのデータ・ローを処理します。

```
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 句

出力ストリームに **MATCHING** 句を指定します。

この例では、スキーマ `TradeSchema` を作成してから、2 つの入力ウィンドウ `InTrades` と `InTrades2`、出力ウィンドウ `TradeOut` を作成します。これらのウィンドウは、それぞれ `TradeSchema` を参照します。

`TradeOut` は、**MATCHING** 句を使用して、一致するローを 1 秒間の間隔で取得します。

```
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
    ;
```

この例では、File CSV Input アダプタ `csvInConn1` を `InTrades` に付加し、File CSV Input アダプタ `csvInConn2` を `InTrades2` に付加します。 また、この例では、File CSV Output アダプタ `csvOut` を `TradeOut` に付加し、一致する結果をファイルにパブリッシュします。これは、データをストリーム内で表示できないためです。

連続したイベントの照合

出力ストリームに **MATCHING** 句と **WHERE** 句を指定して、連続したデータのセットを生成します。

この例では、3 つのスキーマ (`StocksSchema`、`OptionsSchema`、`OutSchema`) を作成します。 この例では次に、`StocksSchema` を参照する入力ウィンドウ `InTrades`、`OptionsSchema` を参照する入力ウィンドウ `InOptions`、`OutSchema` を参照する 2 つの出力ストリーム `TradeOptMatch` と `TradeOptFilter` を作成します。

TradeOptMatch は、**MATCHING** 句を使用して、一致および同じ取引の記号を持つローを 1 秒間の間隔で取得します。TradeOptFilter は、**SELECT** 文を使用して TradeOptMatch からデータを引き出します。**WHERE** 句は、0.005 * TradeOptMatch.StockPrice の積がオプション価格より大きい場合にのみ TradeOptMatch からデータを出力するように TradeOptFilter に指示します。

```
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

;
```

この例では、File CSV Input アダプタ csvInConn1 を InTrades に付加し、File CSV Input アダプタ csvInConn2 を InOptions に付加します。 また、この例では、File CSV Output アダプタ outAdapter を TradeOptFilter に付加し、フィルタした結果をファイルにパブリッシュします。これは、データをストリーム内で表示できないためです。

イベント以外との照合

出力ストリームに **MATCHING** 句と **!** (それ以外) 条件を指定します。

この例では、スキーマ `TradeSchema` を作成してから、入力ウィンドウ `InTrades`、出力ストリーム `TradeOut` を作成します。これらのウィンドウは、いずれも `TradeSchema` を参照します。

`TradeOut` は、`MATCHING !` (それ以外と一致) 構文を使用して、取引を行った株のデータを 10 ミリ秒の間隔で 2 回取得しますが、3 回目は取得しません。

```
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
    ;
```

この例では、File CSV Input アダプタ `csvInConn1` を `InTrades` に付加します。また、この例では、File CSV Output アダプタ `csvOut` を `TradeOut` に付加し、一致する結果をファイルにパブリッシュします。これは、データをストリーム内で表示できないためです。

ローの時間

ローの挿入時間を取得するには、`bigdatetime` システム・カラムを使用します。

この例では、スキーマ `TradeSchema` を作成します。

この例では、スキーマ `TradesWidthDelaySchema` を作成し、**INHERITS** 構文を使用して `TradeSchema` の構造をローの遅延が発生した `TradesWidthDelaySchema` に適用します。

この例では、File CSV Input アダプタを付加する対象となる入力ウィンドウ `TradeWindow` を作成します。

この例では次に、TradesWidthDelaySchema で定義された構造を使用する出力ウィンドウ TradesWithDelay を作成します。 **SELECT** 句は、タイムスタンプ、記号、価格、個数のデータのローにローの遅延を発生させます。 **HAVING** 句でローの遅延が 10 ミリ秒と定義されます。 結果は Symbol でグループ分けされます。

```
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
;
```

第 9 章　モジュールの例

Event Stream Processor には、モジュールの作成とロードの例を備えています。

注意： 構文例は、スペースの制約により折り返されることがあります。 折り返された行は、1 行に入力してください。

CREATE MODULE

モジュールを作成し、後で **LOAD MODULE** 文を使用してプロジェクトに追加できます。

この例では、後で **BEGIN-END** ブロックで定義される入力ウィンドウと出力ウィンドウを識別するモジュール `Module1` を作成します。

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

この例では、**BEGIN-END** ブロックで、パラメータ `myparam` を宣言し、デフォルト値を 2 に設定します。また、この例では、メモリ・ストア `store1` を作成します。

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

            CREATE DEFAULT MEMORY STORE store1;
```

この例では、2 つのスキーマ `inputSchema` と `outputSchema` を作成します。 次に、`inputSchema` を参照する入力ウィンドウ `rawStockFeed` と、`outputSchema` を参照する出力ウィンドウ `infoByStockSymbol` を作成します。 後で文で参照される関数 **getRecordCount()** が **DECLARE** ブロックを使用して宣言されます。

出力ウィンドウ `infoByStockSymbol` は、**SELECT** 句と **FROM** 句を使用して `rawStockFeed` からデータを引き出します。 **WHERE** 句は、株式数が `myparam` に設定されている値より大きい場合に、データにフィルタを設定します。 この例は、**BEGIN-END** ブロックを閉じることによって完了します。

```
                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;
```

モジュールのロード

モジュールをインポートしてロードします。

この例は、**IMPORT** 文を使用して **CREATE MODULE** の例で定義されたモジュールをロードします。このモジュールは `module1.ccl` として保存されます。

この例では、**IMPORT** 文を使用して `module1.ccl` をロードします。

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

この例では、2 つのスキーマ `StocksSchema` と `ComputedStocksSchema`、およびデフォルト・ストア `MyStore1` とメモリ・ストア `MyStore2` を作成します。

この例では次に、`StocksSchema` を参照する入力ウィンドウ `InStocks` を作成します。この入力ウィンドウに File CSV Input アダプタ `csvInStocks` を付加します。

この例では、**LOAD MODULE** 文を使用して、`Module1` のロード、モジュール内で識別された入力ウィンドウの `InStocks` へのリンク、および `MyStore1` の参照が行われます。 この例では、新しい出力ウィンドウは作成されませんが、`Module1` からロードしたウィンドウに新しい名前 (`CompStocks2`) が割り当てられます。また、この例では、`Module1` で宣言された `myparam` パラメータの値を設定します。

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

この例では、ComputedStocksSchema を参照する出力ウィンドウ myw2 を作成します。**SELECT *** (すべてを選択) 構文は、CompStocks2 で処理されたすべてのデータを myw2 に出力します。

第 10 章　高度な例

Event Stream Processor には、さまざまな CCL 要素を含む、高度なプログラミングの例を備えています。

注意： 構文例は、スペースの制約により折り返されることがあります。 折り返された行は、1 行に入力してください。

ポートフォリオ評価

株式ポートフォリオの出来高加重平均価格を計算します。

この例では、入力ウィンドウ `PriceFeed` と出力ウィンドウ `VWAP` を作成します。`VWAP` は、`PriceFeed` によって処理される取引値の出来高加重平均価格の結果を出力します。 結果は `Symbol` でグループ分けされます。 **cast** 関数は株価を `float` に変換します。

```
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;
```

この例では、入力ウィンドウ `Positions` と出力ウィンドウ `IndividualPositions` を作成します。`IndividualPositions` は、`Positions` と `VWAP` の記号の値を使用して、これらウィンドウ間のジョインを作成します。

```
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;
```

この例では、出力ウィンドウ ValueByBook を作成します。この出力ウィンドウは、**SELECT** 句と **FROM** 句を使用して、帳簿 ID の値を基に IndividualPositions からデータを引き出します。ValueByBook は帳簿 ID でそのデータをグループ分けします。

```
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;
```

この例は、File XML Input アダプタ Adapter1 を PriceFeed に付加し、File XML Input アダプタ Adapter2 を Positions に付加することによって完了します。

取引ログ

フレックス・ストリームを使用してウィンドウから手動でデータを削除します。

この例では、**MEMORY** ストア store1 を作成してから、store1 を参照する 2 つの入力ウィンドウ Trades と Trades_truncate を作成します。

この例では、File CSV Input アダプタ Adapter1 を Trades に付加します。 このアダプタは、exampledata フォルダ内のファイル pstrades1.xml からサンプル・データを読み取り、その情報を Trades にパブリッシュします。

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

この例では、Trades と Trades_truncate で動作するフレックス文 Ccl_2_Trades_log を作成します。これは、出力ウィンドウ Trades_log を生成します。 この例では、フレックス文で **DECLARE** ブロックを使用して、2 つの longs を宣言し、この例でこれまでに生成した最小のシーケンス番号と最大のシーケンス番号を格納します。

```
CREATE FLEX Ccl_2_Trades_log
 IN Trades, Trades_truncate
 OUT  OUTPUT   WINDOW Trades_log
     SCHEMA (sequenceNumber LONG, opcode INTEGER, Id INTEGER,
         Symbol STRING, TradeTime DATE, Shares INTEGER, Price
```

```
MONEY(4))
    PRIMARY KEY (sequenceNumber)
     STORE store1
BEGIN
DECLARE
    LONG low;
    LONG high;

END;
```

ON 句は、Trades ウィンドウでレコードが生成されるときは、常に以下のコードを実行します。 このレコードがフレックス・ストリームで最初に表示されるレコードである場合、一連の **if**、**else**、**while** の条件は、最大のシーケンス番号と最小のシーケンス番号を初期化するようにプロジェクト・サーバに指示します。 この例では、反復子を使用して Trades_log 内のすべてのレコードをスキャンし、ログに格納された最小のシーケンス番号と最大のシーケンス番号を検出します。 この例で、Trades_log の反復処理が終了すると、ログに存在する最大のシーケンス番号と最小のシーケンス番号は格納され、反復子は削除されます。

```
ON Trades {
    {
        LONG sn;
        if ((high is null))
            {
                Trades_log_iterator:=getIterator(Trades_log_stream);
                Trades_log:=getNext(Trades_log_iterator);
                if ( not ((Trades_log is null)))
                    {
                        high:=cast(LONG ,0);
                        low:=9223372036854775807;
                    }
                else
                    {
                        high:=cast(LONG ,-1);
                        low:=cast(LONG ,0);
                    }
                while ( not ((Trades_log is null)))
                {
                    sn:=Trades_log.sequenceNumber;
                    if ((sn> high))
                        {
                            high:=sn;
                        }
                    if ((sn< low))
                        {
                            low:=sn;
                        }
                    Trades_log:=getNext(Trades_log_iterator);
                }
                deleteIterator(Trades_log_iterator);
            }
```

この例では、最大のシーケンス番号を 1 ずつ増加し、このシーケンス番号を現在進行中の取引に割り当てます。 最初のレコードでは、シーケンス番号は 0 です。

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


 };
```

ON 句は、`Trades_truncate` ウィンドウでレコードが生成されるときは、常に以下のコードを実行します。

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

一連の **if** と **while** の条件は出力のフォーマットを指定します。 この例では、`Trades_truncate` で指定されたシーケンス番号を取得します。 この番号より小さいシーケンス番号を持つすべてのレコードは、取引ログから削除されます。 要求されたシーケンス番号が取引ログ内の最大のシーケンス番号以上である場合、この例では、取引ログの最新のレコードを除くすべてを削除します。

```
 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))
                        {
```

この例では、指定された値より少ないシーケンス番号それぞれに、opcode 13 (**SAFE DELETE**) を持つレコードを作成します。 安全な削除とは、すべての後続のウィンドウからレコードが削除され (存在する場合)、存在しない場合はエラーが発生しないことを意味します。

```
                                    outrec:=[sequenceNumber=low; |
opcode=cast(INTEGER ,null);
                                  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;
```

索引

D

あ

す

て

は

ふ

も