Tutorial: Solution Patterns for Realtime Streaming Analytics

Srinath Perera, Sriskandarajah Suhothayan
WSO2 Inc.
Mountain View, CA, USA
{srinath, suho}@wso2.com

ABSTRACT

Large-scale data analytics has received much attention under the theme "Big Data". Big data usecases have found a wide range of applications from individual health monitoring to urban planning. Even at this initial stage, big data has demonstrated it's potential to transform the world. Although most early use cases used batch processing technologies like MapReduce, there are many usecases such as stock markets, traffic, surveillance, and patient monitoring that need realtime analytics. Realtime Analytics Technologies like Apache Storm, Spark Streaming, and several Complex Event Processing systems have received attention under realtime analytics. However, most practitioners still focus on implementing realtime analytics from the scratch. There is no common shared understanding about how to implement those analytics usecases among the early adopters. This tutorial tries to address this gap by describing thirteen common relatime analytics patterns and explaining how to implement them. In the discussion, we will draw heavily from real life usecases done under Complex Event Processing and other technologies.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; I.5 [**Pattern Recognition**]: Implementation

General Terms

Performance, Design

Keywords

Complex Event Processing, Events, Data Processing

1. INTRODUCTION

Google's MapReduce paper published in 2005 introduced a simple and scalable data processing technique that can be used with commodity hardware. Since then, Data Analytics–processing data and deriving actionable insights from data–has received much attention from both the Industry and the Academia. Most initial use cases were batch analytics, which stored data in a disk and then periodically processes data by walking through it. Unless data is small, batch analytics would take minutes to produce an output. However, for some use cases (e.g., stock markets, traffic, surveillance, and patient monitoring), the value of insights degrades very quickly with time. For an example, analytics from stock markets are often useless within milliseconds. Batch processing technologies like MapReduce, which take from minutes to hours to produce an output, are not suitable for such use cases. To fill that gap, architects have adopted technologies like In-Memory Computing (VoltDB [12], ZAP Hana [9]), Stream Processing (e.g., Apache Storm [2], Apache Samza [1]), and Complex Event Processing (Esper [5], WSO2 CEP [20], Tibco StreamBase [11]).

Realtime analytics uses cases have two flavors.

1. Realtime Interactive/Ad-hoc Analytics - users issue ad-hoc dynamic queries and the system responds interactively. Among the examples of such systems are Druid, SAP Hana, VoltDB, MemSQL, and Apache Drill.

2. Realtime Streaming Analytics - users issue static queries once and they do not change. The system process data as they come in without storing). CEP and Stream Processing technologies are two example technologies that enable streaming analytics.

Realtime Interactive Analytics allows users to explore a large data set by issuing interactive queries. They should respond within 10 seconds, which is considered the upper bound for acceptable human interaction. In contrast, this tutorial focuses on Realtime Streaming Analytics, which is processing data and reacting to them very fast as they come in, often within few milliseconds. Such technologies are not new. History goes back to Active Databases (2000+), Stream processing (e.g. Aurora [15], Borealis [14] (2005+) and later Apache Storm), Distributed Streaming Operators as a Database research topic around 2005, and Complex Event processing (see [3]).

However, when thinking about realtime analytics, many think only counting use cases. As we shall discuss, counting use cases are only the tip of the iceberg of real life realtime use cases. Since the input data arrives as a data stream,

DEBS'15, June 29 - July 3, 2015, OSLO, Norway.
ACM 978-1-4503-3286-6/15/06 ...$15.00.
http://dx.doi.org/10.1145/2675743.2774214.

a time dimension always presents in the data. This time dimension enable many powerful use cases.

Under the theme large scale streaming analytics, stream processing technologies like Apache Samza and Apache Storm have received much attention. However, these tools force every programmer to design and implement realtime analytics processing from first principles. For an example, if users need a time window, they need to implement it from first principles. This is like every programmer implementing his own list data structure. Better understanding of common patterns will let us understand the domain better and build tools that handle those scenarios. This tutorial tries to address this gap by describing thirteen common relatime streaming analytics patterns and how to implement them. In the discussion, we will draw heavily from real life use cases done under Complex Event Processing and other technologies.

The next section describes the patterns, and the following section compares them against the patterns identified elsewhere. The fourth section describes pattern implementations. The final section concludes the discussion.

2. REALTIME STREAMING ANALYTICS PATTERNS

Let's first agree on the terminology. Realtime Streaming Analytics accepts one or more data streams as input and produces one or more data streams as output. Each data stream consists of many events ordered in time. Each event has many attributes, but all events in a same stream have the same set of attributes or schema. One could think of data streams as a relational table that never ends where each event is a record in the table.

2.1 Pattern 1: Preprocessing

Preprocessing is often done as a projection from one data stream to the other or through filtering. Potential operations include

1. Filtering some events
2. Reshaping a stream by removing, renaming, or adding new attributes to events in the stream
3. Splitting and combining attributes in a stream
4. Transforming attributes

For example, from a twitter data stream, we might choose to extract the fields: author, timestamp, location, and then filter them based on the location of the author.

2.2 Pattern 2: Alerts and Thresholds

This pattern detects a condition and generates alerts based on a condition. (e.g., Alarm on high temperature). These alerts can be based on a simple value or more complex conditions such as rate of increase etc.

For an example, in a use case we implemented using transit data from London city, we trigger a speed alert when a bus has exceeded a given speed limit. More details are described in TFL (Transport for London) Demo video [13].

We can also generate alerts for more complex scenarios such as detecting if the server room temperature is continually increasing for the last five mins.

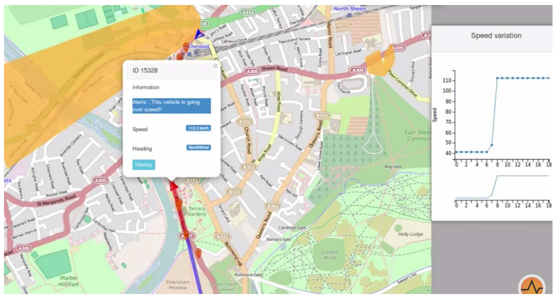

Figure 1: TFL Demo

2.3 Pattern 3: Simple Counting and Counting with Windows

This pattern includes aggregate functions like Min, Max, Percentiles, etc. They can be calculated without storing any data. (e.g., counting the number of failed transactions).

However, counts are often useful with a time window attached to it.(e.g. failure count in the last hour). There are many types of windows: sliding windows vs. batch (tumbling) windows and time vs. length windows. There are four main variations.

1. Time-sliding window: keeps each event for the given time window, produces an output whenever a new event has been added or removed.
2. Time-batch window (called tumbling window): produces output at the end of the given time window
3. Length-sliding : same as the time-sliding window, but keeps a window of n events instead of selecting them by time.
4. Length-Batch window: same as the time-batch window, but keeps a window of n events instead of selecting them by time

Also, there are special windows like decaying windows and unique windows.

2.4 Pattern 4: Joining Event Streams

The main idea behind this pattern is to combine multiple data streams and create a new event stream. For example, let's assume we play a football game where both the players and the ball having attached sensors that emit events that have current location and acceleration as properties. We can use "joins" pattern to detect when a player has kicked the ball. We do that by joining the ball location stream and the player stream on the condition that they are close to each other by one meter and the ball's acceleration has increased by more than $55m/s^2$.

Among other use cases are combining data from two sensors, detecting collisions, and detecting proximity of two vehicles.

2.5 Pattern 5: Data Correlation, Missing Events, and Erroneous Data

When we correlate events between different streams, we would use the forth pattern. In addition, we can also correlate the data within the same stream. Another variation of this idea is missing events and erroneous data.

Following are some of the possible scenarios.

1. Matching up two data streams that send events in different speeds
2. Detecting a missing event in a data stream (e.g., detect a customer request that has not been responded within 1 hour of its reception.)
3. Detecting erroneous data (e.g., Detecting failed sensors using a set of sensors that monitor overlapping regions. We can use those redundant data to find erroneous sensors and removing their data from further processing)

2.6 Pattern 6: Interacting with Databases

Often we need to combine the realtime data against the historical data stored in a disk. Following are few examples.

1. When a transaction happened, lookup the age using the customer ID from customer database to be used for fraud detection (enrichment)
2. Checking a transaction against blacklists and whitelists in the database
3. Receive an input from the user (e.g., Daily discount amount may be updated in the database, and then the query will pick it automatically without human intervention).

2.7 Pattern 7: Detecting Temporal Event Sequence Patterns

Using regular expressions with strings, we detect a pattern of characters from a sequence of characters. Similarly, given a sequence of events, we can write a regular expression to detect a temporal sequence of events arranged on time where each event or condition about the event is parallel to a character in a string in the above example.

Often cited example, although bit simplistic, is that a thief, having stolen a credit card, would try a smaller transaction to make sure it works and then do a large transaction. Here the small transaction followed by a large transaction is a temporal sequence of events arranged on time, and can be detected using a regular expression written on top of an event sequence.

Such temporal sequence patterns are very powerful. For example, the [7] shows a real time analytics done using the data collected from a real football game. This was the data set taken from the DEBS 2013 Grand Challenge.

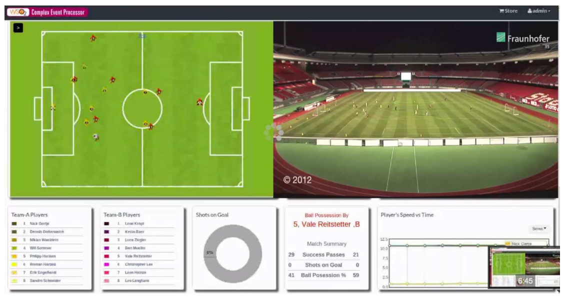

Figure 2: Football Demo

In the video, we used event sequence patterns to detect the ball possession (the time period a specific player controlled the ball). A player possessed the ball from the time he hits the ball, until someone else hits the ball. This condition can be written as a regular expression: a hit by me, followed by any number of hits by me, followed by a hit by someone else. (We already discussed how to detect the hits on the ball in Pattern 4: Joins).

2.8 Pattern 8: Tracking

The eighth pattern tracks something over space and time and detects given conditions. Following are few examples

1. Tracking a fleet of vehicles, making sure that they adhere to speed limits, routes, and geo-fences.
2. Tracking wildlife, making sure they are alive (they will not move if they are dead) and making sure they will not go out of the reservation.
3. Tracking airline luggage and making sure they have not been sent to wrong destinations
4. Tracking a logistic network and figuring out bottlenecks and unexpected conditions.

For example, TFL Demo we discussed under pattern 2 shows an application that tracks and monitors London buses using the open data feeds exposed by TFL (Transport for London).

2.9 Pattern 9: Detecting Trends

We often encounter time series data. Detecting patterns from time series data and bringing them into operator attention are common use cases.

Following are some of the examples of trends.

1. Rise, Fall of values
2. Turn (switch from rise to a fall)
3. Outliers - deviate from the current trend by a large value
4. Complex trends like “Triple Bottom” and “Cup and Handle” [17].

These trends are useful in a wide variety of use cases such as

1. Stock markets and Algorithmic trading
2. Enforcing SLA (Service Level Agreement), Auto Scaling, and Load Balancing
3. Predictive maintenance (e.g., guessing whether the Hard Disk will fill within the next week)

2.10 Pattern 10: Running the Same Query in Batch and Realtime Pipelines

This pattern runs the same query in both Relatime and batch pipelines. It is often used to fill the gap left in the data due to batch processing. For example, if batch processing takes 15 minutes, results would always lags 15 minutes from the current data.

The idea of this pattern, which is sometimes called “Lambda Architecture”, is to use realtime analytics to fill the gap. Nathen Marz’s “Questioning the Lambda Architecture” [8] discusses this pattern in detail.

2.11 Pattern 11: Detecting and Switching to Detailed Analysis

The main idea of the pattern is to detect a condition that suggests some anomaly, and further analyze it using historical data. This pattern is used with the use cases where we cannot analyse all the data in full detail. Instead, we analyze only anomalous cases in full detail. Following are few examples.

1. Use basic rules to detect Fraud (e.g., large transaction), then pull out all transactions done against that credit card for a larger time period (e.g., 3 months data) from batch pipeline and run a detailed analysis
2. While monitoring weather, detect conditions like high temperature or low pressure in a given region, and then start a high resolution localized forecast for that region.
3. Detect good customers, for example through expenditure of more than $1000 within a month, and then run a detailed model to decide the potential of offering a deal.

2.12 Pattern 12: Using a Model

The idea is to train a model (often a Machine Learning model), and then use it with the Realtime pipeline to make decisions. For example, you can build a model using R, export it as PMML (Predictive Model Markup Language) and use it within your realtime pipeline.

Following are few examples.

1. Fraud Detection
2. Segmentation
3. Predict the next value
4. Predict Churn

2.13 Pattern 13: Online Control

There are many use cases where we need to control something online. The classical use cases are autopilot, self-driving, and robotics. These would involve problems like current situation awareness, predicting next value(s), and deciding on corrective actions.

3. RELATED WORK

Earlier efforts have identified common streaming realtime operators and patterns. For an instance, most Complex Event Processing technologies have adopted a SQL like query languages thus adopting operators from SQL. They, however, added more operators like windows and temporal pattern recognition to better handle the time dimension. Most such tools support patterns such as filters, transforms, windows, joins, and temporal event patterns. Cugola and Margara [18] have defined a classification of these operators and done an extensive survey of different tools. However, as shown in Table 1, the paper focuses on basic abstract operators as oppose to higher-level patterns.

The technical community has also discussed this issue several times. For an example, Esper provides about 50 patterns in their solution patterns document [6]. Although the document does not try to cover all categories, it provides insight into common problems faced by streaming query developers. Most Esper patterns are very specific and defined on narrow use cases. However, since the patterns have grown from user feedback over time, they provide a good indication of problems people trying to solve with event processing systems.

Coral8 white paper [4] discusses ten patterns. As Table 1 depicts most of them matche directly to the patterns discussed in this tutorial. Among ones that did not match, we considered in-memory caching as an implementation detail of database mapping (Pattern 6). We considered hierarchical processing as a preprocessing step in generating events. Also we considered dynamic queries not as a pattern, but as an implementation detail about how users submit queries to the system.

Event Processing Technical Society's (EPTS) reference architecture [19] comes very close to the patterns we discussed in this tutorial. In contrast to other efforts, the EPTS reference architecture identifies higher-level patterns such as tracking, prediction and learning in addition to low-level operators that comes from SQL like languages. Allowing for wider interpretations, EPTS patterns can contain almost all patterns we discuss. A notable exception is streaming patterns that incorporate batch processing, which is an idea developed much later.

The patterns we introduced incorporate basic complex event processing patterns that originated from SQL like languages as well as the higher-level use cases like tracking and detecting trends. They are defined with more focus on the user rather than with the mechanics of processing implementation. For example, we called the first two patterns "Preprocessing and Alerts" instead of calling it a filter. Following Table 1 shows how each of above approaches matches the patterns introduced in this tutorial.

4. PATTERN IMPLEMENTATIONS

In this section we will discuss how the above patterns can be implemented with Apache Storm [2] and WSO2 Complex Event Processor [20].

Apache Storm is a distributed stream processing engine, which allows users to write queries (topologies as per Apache Storm terminology) as JavaBeans and deploy them into a cluster. It uses components named Spouts for fetching events from external systems and Bolts for processing those events. Since Spouts and Bolts are written as Java classes it allows us to implement almost any use case. In this tutorial we are omitting the implementation details of Spouts because it is same for all the use cases and depends on the transports and the protocol used for retrieving the events.

As the Complex Event Processing Server, we will use WSO2 CEP. It accepts queries via SQL like query language and supports filters, transformations, windows, joins, temporal event patterns, and interacting with databases with data streams. At the distributed mode, WSO2 CEP can compile a given query to set of queries that run as an Apache Storm topology where each Bolt would run a WSO2 CEP engine as a Java library. We call this Java Library as Siddhi [10].

Following is an example query supported by WSO2 CEP.

```
from inputStockStream[price > 100]
select symbol, price * volume as transactionAmount
insert into outputStockStream
```

Table 1: Comparison With Earlier Methods

	Coral8 patterns	Esper	Cugola	EPTS Reference Arch
Pattern 1: Preprocessing	yes	yes	yes	yes
Pattern 2: Alerts and Thresholds		yes		
Pattern 3: Simple counting and Counting with Windows	yes	yes	yes	yes
Pattern 4: Joins	yes	yes	yes	yes
Pattern 5: Data Correlation, Missing and Erroneous Data		yes	yes	yes
Pattern 6: Interacting with Databases	yes			
Pattern 7: Detecting Temporal Event Sequence Patterns	yes	yes	yes	yes
Pattern 8: Tracking				yes
Pattern 9: Detect trends		yes		yes
Pattern 10: Same Query in Batch and Realtime Pipelines				
Pattern 11: Detecting and switching to Detailed Analysis				
Pattern 12: Using a Model				yes
Pattern 13: Online Control				

Similar to Apache Storm, for simplicity, we are going to omit the implementation details of event retrieval and stream management configurations with WSO2 CEP as well.

Apache Storm based implementation of the above described patterns needs writing Java code. For brevity, we will use pseudocode to explain the implementations.

Each Apache Storm query is specified as one or more bolts. Each bolt is represented as a method called PROCESS(E) where E is the event. E.attribute-name refers to a specific attribute of the event. Method NEW_E() creates a new event. Also there is a special method SEND2_NEXT_PROCESSOR(E) that sends the event to next processor (bolt) in the topology. We will also use few utility methods and make their function obvious by the name. Apache Storm decides next processor based on the topology.

While explaining the WSO2 CEP based solution, we will use Siddhi Query language, which is the SQL like event query language used by WSO2 CEP.

4.1 Implementation of Preprocessing

For the first pattern, let's consider a StockQuote Stream with attributes transactionAmount, symbol, price, volume, and a timeout. In this example, we filter all transactions greater than 100 and then transform the stream to output only the symbol and the last transactionAmount attributes of that stock.

The pseudo code implementation of filtering events can be depicted as below.

```
PROCESS(E)
  if E.price >= 1000
    SEND2-NEXT(E);
  END
END
```

Similarly reshaping, transforming streams, splitting and combining the attributes of a stream can be implemented as follows.

```
PROCESS(E, N)
  En = NEW_E();
  //transform
  En.transactionAmount = E.price*E.volume;
  En.symbol = E.symbol;
  SEND2_NEXT_PROCESSOR(En);
 END
```

Following CEP query implements both filtering and transformations.

```
from inputStockStream[price > 100]
select symbol, price * volume as transactionAmount
insert into outputStockStream
```

4.2 Implementation of Alerts and Thresholds

This pattern can be implemented using the same techniques as filters in the first pattern. We have identified Alerts as a different pattern because the user's intent is different in the two patterns.

4.3 Implementing Simple Counting and Counting with Windows

The implementation of counting can be explained by an example where we collect number of events and then calculate aggregate values based on those collected events. For an example, we can use this pattern to calculate the sum of all stock quotes last 60 seconds.

For Storm implementation we assume a window data structure that collects data and can detect when a window has expired.

```
W = NEW WINDOW(WINDOW_TIME);

PROCESS(E)
  WINDOW_ADD(E.price*E.amount, E.timeStamp);
  REMOVE_EXPIRED_EVENTS(W, E.timestamp)
  FOR v in W
    SUM = SUM + V;
  END
  En = NEW_E();
  En.sumOfAmount =  SUM;
  SEND2_NEXT_PROCESSOR(En);

END
```

We can implement a counting example with CEP as follows.

```
from inputStockStream[symbol == 'GOOG']#window.time(60s)
select sum(price*volume) as totalAmount
insert into lastMinStocksOfGOOG;
```

Here we are implementing a 60 seconds batch window and we use it to generate the totalAmount of the GOOG stocks that were trading over the last hour. However, there are many other types of windows available: sliding time windows, length batch window, length sliding window, unique window, first unique window, etc. We will not discuss them in detail, but most Complex Event Processing engines, including WSO2 CEP, have support for those windows.

4.4 Implementation of Event Streams Joins

When joining multiple streams, we can not join the current event of a particular stream against all the events that have arrived so far from the other stream. This is because it is expensive to hold all the events in memory or even at an external storage. Therefore, with realtime streaming use cases, we store only a predefined amount of events (based on time, length or via other criteria) and match against those. For a successful implementation of the join pattern, we will need a criteria for storing historic events and the ability to match the conditions against that dynamic storage.

For this example, let's consider the football game described with the pattern where we join the ball and player streams to detect a kick. The following pseudo code explains the Storm implementation.

For this use case, we assume there are two window implementations that can be used to retain events.

```
BALL_W = NEW_LENGTH_WINDOW(1);
PLAYER_W = NEW_UNIQUE_WINDOW();

PROCESS(E)
  if E instanceof  ballStream
    UPDATE_WINDOW(BALL_W, E);
    IF E.acceleration > 55
        LIST = FIND_EVENTS(PLAYER_W, E, "I ∈ W
            | mod(I.position - " + E.position + " < 1");
        for Et in LIST
          SEND2_NEXT_PROCESSOR(Et);
        END
    END
  else
    UPDATE_WINDOW(PLAYER_W, E)
    LIST = FIND_EVENTS(BALL_W, E,
      "I ∈ W| I.acceleration > 55
      and mod(I.position - " + E.position );
    for Et in LIST
      SEND2_NEXT_PROCESSOR(Et);
    END
  END
END
```

This implementation keeps two windows, one for events from the ball stream and one for the events from player stream. Whenever, we receive an event we update the corresponding window and search for a matching event in the other window based on the given condition. Any events that matches we send forward to the next processor.

If windows could hold a large number of events, we would need to build indexes to make FIND_EVENTS faster.

Following query shows the same use case implemented with Complex Event Processing.

```
from ballStream#window.length(1)
  join palyerStream#window.unique(palyerId)
  on ballStream.acceleration > 55
    and mod(ballStream.position -
        palyerStream.position) < 1
select ballStream.position, palyerStream.palyerId,
insert into ballKickSteam;
```

4.5 Implementation of Data Correlation, Missing Events and Erroneous Data

We can implement data correlation using the same techniques as Pattern 4. However, detecting missing events and erroneous data is much more complex. Here we will look at how we can implement detection of missing events in a data stream. As the example use case, we detect the customer requests that have not been responded within 1 hour of it's reception.

```
W = NEW_SLIDING_WINDW(1h);

PROCESS(E)
    if E instanceof RequestEvent
       EX_LIST = PUT_AND_GET_EXPIRED_EVENTS(E, E.timeStamp);
       for En in EX_LIST
           SEND2_NEXT_PROCESSOR(En);
      END
    else if E instanceof ResponseEvent
       REMOVE_EVENT("e ∈ W| E.txID == e.txID")
    else
       ERROR();
    END
END
```

Here we keep a list of requests and remove them when the request has been served (detected via ResponseEvent). If the request has stayed in the list for more than 1 hour, we raise an event after it is expired.

CEP query for the same use case is given below.

```
from requestStream#time.window(1 hour)
insert expired events into expiryStream;

from r1=requestStream ->
  r2=responceStream[r1.id == id]
  or r3=expiryStream[r1.id == id]
select r1.id as id, ...
insert into alertStream
having r2.id == null
```

4.6 Implementation of Databases Interactions

We can implement Pattern 6 with storm using the code similar to Pattern 4 with calls to a database instead of storing and searching for data in-memory. Furthermore, it will only have to keep events for one stream.

For example, let's assume we want to retrieve the customer age from a database using customer ID attribute as the key and enrich the event with customer's age.

```
PROCESS(E)
  RECORD = READ_FROM_DB_WITH_CACHE(
    "SELECT age FROM CUSTOMER_TABLE
```

```
      WHERE custID = "+ E['custID']");
   E.age = RECORD.age;
   SEND2_NEXT_PROCESSOR(E);
END
```

In WSO2 CEP, we use a construct called "Event Table" for merging events with data in a database and for collecting and conditionally updating data in the database. This allows Siddhi to represent historical data stored in a database as a window allowing it to be combined with active event streams in realtime.

Let us implement the earlier use cases with CEP.

```
define table CustomerTable (customerId string,
  customerName string, customerAge int);

from TxStream join CustomerTable
  on TxStream.userId == CustomerTable.customerId
select userId as id, CustomerTable.customerAge as age,
insert into fraudDetectionStream
```

Here event table matches data in a database to a window, and we can join against the window as any window.

4.7 Implementation of Temporal Event Sequences and Patterns

Implementing sequences and patterns with Storm is complex. Hence let us look at a simple fraud detection use case where we detect a small transaction (an amount less than $50) from a credit card, followed by a very large transaction (an amount larger than $10000) from the same card within a day.

The pseudo code explaining the scenario implementation is as follows.

```
M = NEW_MAP()

PROCESS(E)
  KEY = CONCAT(E.cardNumber);
  if E.amount < 50
    MAP_PUT(M, KEY , E.amount);
  else If E.amount > 10000
    if MAP_CONTAINS(M, KEY)
      MAP_REMOVE(M, KEY));
      //send the event
      SEND2_NEXT_PROCESSOR(E);
    END
  else
    ERROR();
  END
END
```

The above code detects two events that follow each other. In the general case, the pattern can be expressed as a regular expression and the system needs to simulate a state machine to implement them. Moreover, the system needs to expire events to avoid pattern collecting all small transaction events and running out of memory. We will not discuss the implementation of these details in this paper.

At the meantime temporal sequences and patterns can be easily implemented in CEP systems as they are provided as an out of the box constructs. The following segment of Siddhi query explains the above scenario.

```
from a1 = transactionStream[amount < 50 ] ->
        a2 = transactionStream[a1.cardNo == cardNo
          and amount > 10000]
        within 1 day
select a1.cardNo as cardNo, a1.amount as firstAmmount,
  a2.amount as currentAmount, ...
insert into fraudDetectionStream
```

Here '->' denotes 'followed by', meaning after the first event its corresponding next event can appear after any number of other events.

4.8 Implementation of Trend Detection

Detecting trends is one of the key streaming analytics use cases. Trends are widely used in both Stock markets and algorithmic trading. For example, stock market traders have identified chart patterns like Triple Bottom pattern [17] that are significant behaviors in the market that are actionable.

Identifying even basic trends like a peak require us to implement a state machine depicted in Figure 5. We will not implement this use case with Storm as it is complicated.

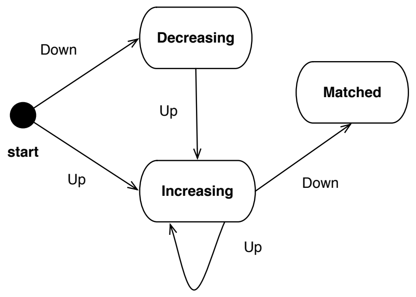

Figure 3: State Machine to Detect a Peak

However, if we ignore the noise, we can easily detect most trends as temporal event patterns using CEP. For example, the following query identifies a peak in GOOG stock price.

```
from stockStream[symbol=='GOOG']
insert into googStockStream

from a1=googStockStream,
        a2=googStockStream[last.price<=price]+,
        a3=googStockStream[a2.price>price]
select a2[last].price as peekPrice,
  a3.price as currentPrice
```

If we need to process the above for all the symbols independently, then we have to partition the stockStream and process the events in parallel and this can be implemented as below.

```
partition with ( symbol of stockStream)
begin
    from a1=stockStream,
            a2=stockStream[last.price<=price]+,
            a3=stockStream[a2.price>price]
    select a2[last].price as peekPrice,
      a3.price as currentPrice
end;
```

However, a real life trend detection needs to be resilient against the noise. For an example, Rashmi Arachchi et al. [16] used kernel regression to detect maxima and minima points and then applied temporal events patterns to those maxima and minima points to identify high-level chart patterns.

4.9 Implementing the Same Query in Batch and Realtime Pipelines

This pattern is often implemented combining a batch-processing tool like MapReduce and a realtime processing tool like CEP. As shown by Figure 4, the main idea is to periodically run the batch processing code and place the results in a database. Then we combine the results from the batch process that is in the database or a distributed cache using Pattern 6: Merge with data in a database.

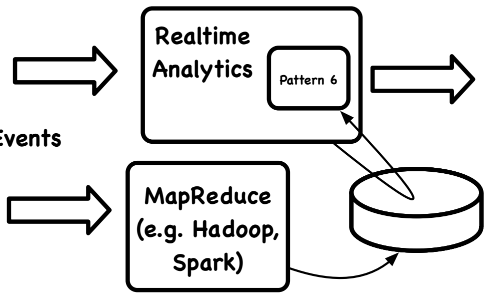

Figure 4: Lambda Architecture

4.10 Implementation of Detecting a Condition and Switching to Detailed Analysis

For an example, we can detect fraudulent conditions using a realtime engine in streaming fashion and then trigger a detailed analysis with a batch processing engine like MapReduce. As shown in Figure 5, we can implement this pattern by first using Alert, Event Sequences, Trends patterns to detect the condition and then triggering a batch job using a MapReduce cluster.

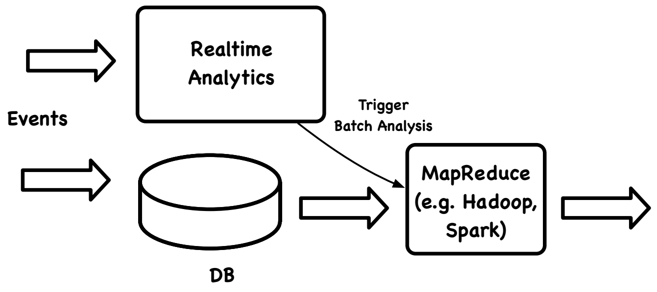

Figure 5: Trigger Batch from Realtime

For example, we have built a credit card fraud detection solution that detects suspicious activities and lets a human operator explore related data for more information. For example, when the system flags a credit card transaction as suspicious, the operator can pull out all other transactions done by the same card and make a final decision. Instead by having a human int the loop, the system can also trigger a detailed analysis.

4.11 Implementation of Using a Model

The idea is to use a Machine Learning model built elsewhere within realtime analytic processing. For example, we can use R language to build a model (e.g., logistic regression model) and then export the model as a Predictive Model Markup Language (PMML) Model. We can run the PMML model from WSO2 CEP using it's PMML extension. Similarly, we can use a PMML interpreter like JPMML within a Storm Bolt to implement a similar solution.

Following sample query shows how to use a machine learning model with WSO2 CEP queries.

```
from TrasnactionStream
  #ml:applyModel('/path/logisticRegressionModel1.xml',
    time stamp, amount, ip)
insert into PotentialFraudsStream;
```

4.12 Implementation of Online Control

For example, let's consider keeping a car's speed close to a given limit. This can be implemented by using a pattern we discussed before to detect the conditions and then carry out an action to control the car.

For an example, if we need to keep the car's speed in 50 kmph, we can have one query that detects when it is over 55 kmph and apply the breaks, and another that detects when it is less than 45 kmph and accelerate the car. Both above queries can be implemented with Alerts Pattern.

However, real life implementations of online control are much more complicated due to factors like it takes time for operations to take effect and we need to wait for one operation to finish before issuing another one. Furthermore, certain conditions might lead to oscillations. For example, a car that repeatedly accelerates and decelerates is not very comfortable to ride on. We will not explore those details in this discussion.

5. CONCLUSION

Although there is much interest in realtime analytics, much focus is on counting use cases. Despite being useful, such use cases are only a small portion of realtime analytics use cases. Since the input data arrives as a data stream, a time dimension always presents in the data. This time dimension allows us to implement and perform many powerful use cases.

Most current tools force every programmer to design and implement realtime analytics processing from first principles. For an example, if users need a time window, they need to implement it from first principals. This is like every programmer implementing his own list data structure. Better understanding of common patterns will let us understand the domain better and build tools that handle those scenarios. This tutorial tried to address this gap by describing thirteen common relatime analytics patterns and how to implement them. Furthermore, we discussed several real life use cases where those patterns are useful.

6. REFERENCES

[1] Apache samza. http://samza.apache.org/. Accessed: 2015-05-02.

[2] Apache storm. https://storm.apache.org/. Accessed: 2015-05-02.

[3] Cep tooling market survey. http://www.complexevents.com/2014/12/03/cep-tooling-market-survey-2014/. Accessed: 2015-05-02.

[4] Complex event processing: Ten design patterns. http://complexevents.com/wp-content/uploads/2007/04/Coral8DesignPatterns.pdf. Accessed: 2015-05-02.

[5] Esper complex event processing engine. http://www.espertech.com/. Accessed: 2015-05-02.

[6] Esper solution patterns. http://www.espertech.com/esper/solution_patterns.php. Accessed: 2015-05-02.

[7] Football demo based on debs 2013 grand challenge. https://www.youtube.com/watch?v=nRI6buQ0NOM. Accessed: 2015-05-02.

[8] Questioning the lambda architecture. http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html. Accessed: 2015-05-02.

[9] Sap hana. http://hana.sap.com/abouthana.html. Accessed: 2015-05-02.

[10] Siddhi complex event processing library. https://github.com/wso2/siddhi. Accessed: 2015-05-02.

[11] Tibco streambase. http://www.tibco.com/products/event-processing/complex-event-processing/streambase-complex-event-processing. Accessed: 2015-05-02.

[12] Voltdb. http://voltdb.com/. Accessed: 2015-05-02.

[13] Wso2 london tfl demo. https://www.youtube.com/watch?v=mjPPbTFAqes. Accessed: 2015-05-02.

[14] D. Abadi, Y. Ahmad, et al. The design of the borealis stream processing engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005), Asilomar, CA*, pages 277–289, 2005.

[15] D. Abadi, D. Carney, et al. Aurora: a data stream management system. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 666–666, 2003.

[16] R. Arachchi, M. Bandara, et al. A complex event processing toolkit for detecting technical chart patterns. *High Performance Big Data and Cloud Computing Workshop (HPBC)*, 2015.

[17] T. N. Bulkowski. *Encyclopedia of chart patterns*, volume 225. John Wiley & Sons, 2011.

[18] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys, 2012*.

[19] A. Paschke and P. Vincent. A reference architecture for event processing. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, page 25. ACM, 2009.

[20] S. Suhothayan, K. Gajasinghe, I. L. Narangoda, S. Chaturanga, S. Perera, and V. Nanayakkara. Siddhi: A second look at complex event processing architectures. In *Gateway Computing Environments Workshop (GCE)*. IEEE, 2011.