ANALYTICS ON BIG FAST DATA USING REAL TIME STREAM DATA PROCESSING ARCHITECTURE

Dibyendu Bhattacharya
Architect-Big Data Analytics
HappiestMinds

Manidipa Mitra
Principal Software Engineer
EMC

Table of Contents

Introduction

We all know the Hadoop like Batch processing system had evolved and matured over past few years for excellent offline data processing platform for Big Data. Hadoop is a high-throughput system which can crunch a huge volume of data using a distributed parallel processing paradigm called MapReduce. But there are many use cases across various domains which require real-time / near real-time response on Big Data for faster decision making. Hadoop is not suitable for those use cases. Credit card fraud analytics, network fault prediction from sensor data, security threat prediction, and so forth need to process real time data stream on the fly to predict if a given transaction is a fraud, if the system is developing a fault, or if there is a security threat in the network. If decisions such as these are not taken in real time, the opportunity to mitigate the damage is lost.

Real-time systems perform analytics on short time windows, i.e. correlating and predicting events streams generated for the last few minutes. Now, for better prediction capabilities, real-time systems often leverage batch processing systems such as Hadoop.

The heart of any prediction system is the Model. There are various Machine Learning algorithms available for different types of prediction systems. Any prediction system will have higher probability of correctness if the Model is built using good training samples. This Model building phase can be done offline. For instance, a credit card fraud prediction system could leverage a model built using previous credit card transaction data over a period of time. Imagine a credit card system for a given credit card provider serving hundreds of thousands of users having millions of transactions data over given period of time; we need a Hadoop-like system to process them. Once built, this model can be fed to a real time system to find if there is any deviation in the real time stream data. If the deviation is beyond a certain threshold, it can be tagged as an anomaly.

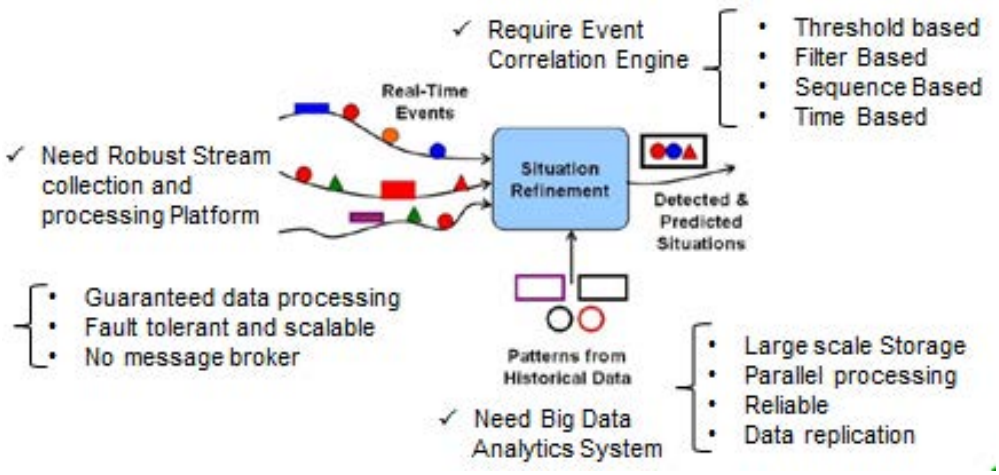


Figure 1

Figure 1 shows the holistic view of the whole system; the Event Stream Collection system, Event Correlation Engine, and a Situation Refinement using models built by a Hadoop- based Big Data system.

In this article we will showcase how real time analytics use cases can be solved using popular open source technologies to process real time data in a fault tolerant and distributed manner. The major challenge here is; it needs to be always available to process real time feeds; it needs to be scalable enough to process hundreds of thousands of message per second; and it needs to support a scale-out distributed architecture to process the stream in parallel.

This article will also describe how we can use Hadoop for building a prediction model for sequence analysis with the help of the Machine Learning library, Mahout (http://mahout.apache.org/), and how this model can be used in the real time system for prediction.

Real Time Data Processing Challenges

Real Time data processing challenges are very complex. As we all know, Big Data is commonly categorized into volume, velocity, and variety of the data, and Hadoop like system handles the Volume and Varity part of it. Along with the volume and variety, the real time system needs to handle the velocity of the data as well. And handling the velocity of Big Data is not an easy task. First, the system should be able to collect the data generated by real time events streams coming in at a rate of millions of events per seconds. Second, it needs to handle the parallel processing of this data as and when it is being collected. Third, it should perform event correlation using a Complex Event Processing engine to extract the meaningful information from this moving stream. These three steps should happen in a fault tolerant and distributed way. The real time system should be a low latency system so that the computation can happen very fast with near real time response capabilities.

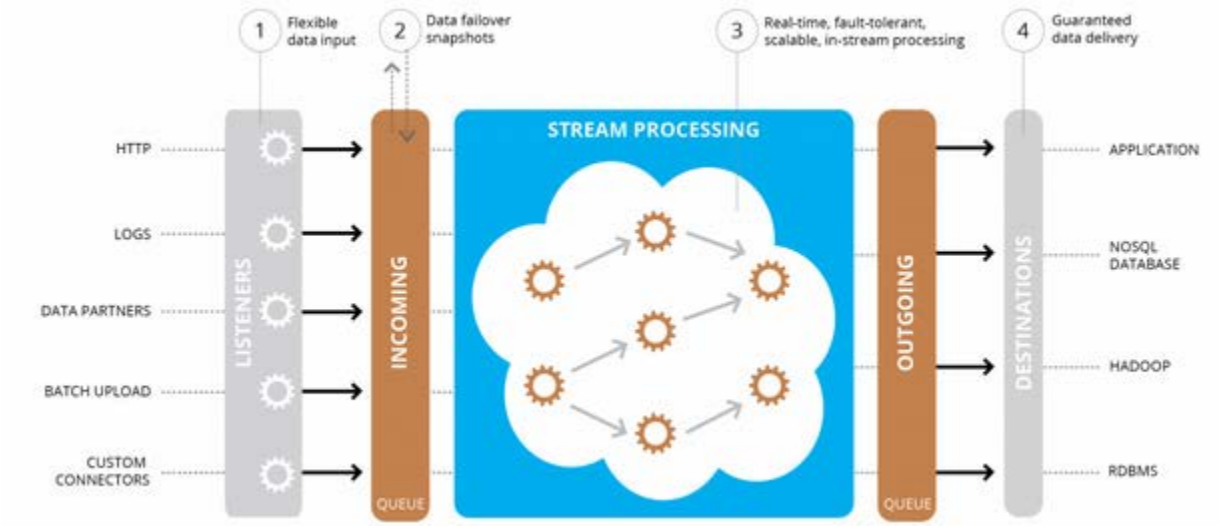


Figure 2

Figure 2 depicts a different construct of a real time system. Streaming data can be collected from various sources, processed in the stream processing engine, and then write the result to destination systems. In between, the Queues are used for storing/buffering the messages.

To solve this complex real time processing challenge, we have evaluated two popular open source technologies; Apache Kafka (http://kafka.apache.org/design.html), which is the distributed messaging system, and Storm (http://storm-project.net/) which is a distributed stream processing engine.

Storm and Kafka are the future of stream processing, and they are already in use at a number of high-profile companies including Groupon, Alibaba, The Weather Channel, and many more.

An idea born inside of *Twitter*, Storm is a "distributed real-time computation system". Meanwhile, Kafka is a messaging system developed at *LinkedIn* to serve as the foundation for their activity stream and the data processing pipeline behind it.

With Storm and Kafka, you can conduct stream processing at linear scale, assured that every message is processed in a real-time, reliable manner. Storm and Kafka can handle data velocities of tens of thousands of messages every second.

Stream processing solutions such as Storm and Kafka have caught the attention of many enterprises due to their superior approach to ETL (extract, transform, and load) and data integration.

Let's take a closer look at Kafka and Storm to how they have achieved the parallelism and robust processing of stream data.

Distributed Messaging Architecture: Apache Kafka

Apache Kafka is the messaging system originally developed at LinkedIn for processing LinkedIn's activity stream. Let look at the issues with the traditional messaging system which caused people to seek an alternative.

Any messaging system has three major components; the message producer, the message consumer, and the message broker. Message producers and message consumers use message queue for asynchronous inter-process communication. Any messaging supports both point-to-point as well as publish/subscribe communication. In point-to-point paradigm, the producer of the messages sends the messages to a queue. There can be multiple consumers associated with the queue, but only one consumer can consume a given message from a queue. In publish/subscribe paradigm, there can be multiple producers producing messages for a given entity called Topic, and there can be multiple consumers subscribed for that topic. Each subscription receives the copy of each message sent for that topic. This differs from point-to-point communication, where one message is consumed by only one consumer. The message broker is the heart of the whole messaging system, which bridges the gap between the producer and consumer.

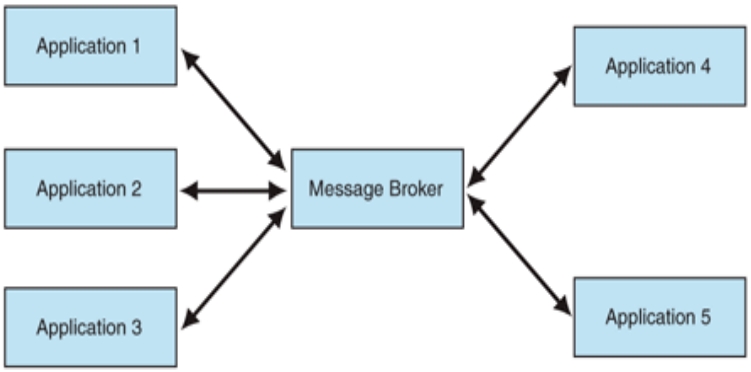


Figure 3

In Figure 3, the applications running either side of the broker can communicate with each other over point-to-point or publish/ subscribe way. The broker acts as a mediation layer between applications which provides some basic messaging properties, such as message durability and message persistency.

A “durable message" is a message where the broker will hold on to a message if the subscriber is temporarily unavailable. So the durability is defined by the relationship between a "Topic Subscriber" and the "Broker". Durability is applicable only to the publish/subscribe paradigm.

A "persistent message" is a message that defines the relationship between a "Message Producer" and the "Broker". This can be established for both point-to-point and publish/subscribe. This has to do with the guaranteed delivery of the message by persisting the message after it has been received from the message producer.

Both message durability and message persistency come with a cost. Keeping the state of the message (whether message is consumed or not) is a tricky problem. The traditional messaging brokers keep track of the consumer state. It uses metadata about the messages and stores this metadata in broker. Storing metadata about billions of messages creates large overhead for broker. On top of that, the relational database for storing the message metadata does not scale very well. However, the broker tries to keep the metadata size small by deleting messages which are already consumed. The challenging problem arises about how Broker and Consumer conclude that a given message is consumed. Can a broker mark a message consumed as and when it put the message in the network for delivery? What will happen if the Consumer is down by the time the message reaches the consumer? To solve this, most messaging systems keep an acknowledgement system. When a message is delivered by a broker, it is marked as “sent”, and when the consumer consumes the message and sends an acknowledgement, the broker marks it as “consumed”. But what will happen if the consumer actually consumed the message,

but a network failure occurred before the acknowledgement reached the broker? The broker will still keep the state as “sent” not “consumed”. If the broker resends the message, the message will be consumed twice. The major problem arises around the performance of the broker. Now, the broker must keep track of each and every message. Imagine the overhead of the broker in cases when thousands of messages are being produced every second. This is a major reason why the traditional messaging system is not able to scale beyond a certain limit.

Another problem for traditional messaging systems is the storage data structure they use. Almost all messaging systems use Relational Database for storing messages. A relational database uses BTree for storing the indexes for faster retrieval. BTrees are the most versatile data structure available, and make it possible to support a wide variety of transactional and non-transactional semantics in the messaging system. But they come with a fairly high cost: BTrees operations are O (log N). Normally O (log N) is considered essentially equivalent to constant time, but this is not true for disk operations. Disk seeks come at 10 ms a pop, and each disk can perform only one seek at a time so parallelism is limited. Hence, even a handful of disk seeks lead to very high overhead. Furthermore, BTrees require a very sophisticated page or row-locking implementation to avoid locking the entire tree on each operation. The implementation must pay a high price for row-locking or else effectively serialize all reads[1].

The following is how Kafka solved these problems.

[1] http://kafka.apache.org/design.html

Messaging System: The Kafka Way

Figure 4 shows how different types of producers can communicate to different types of consumers through the Kafka Broker.

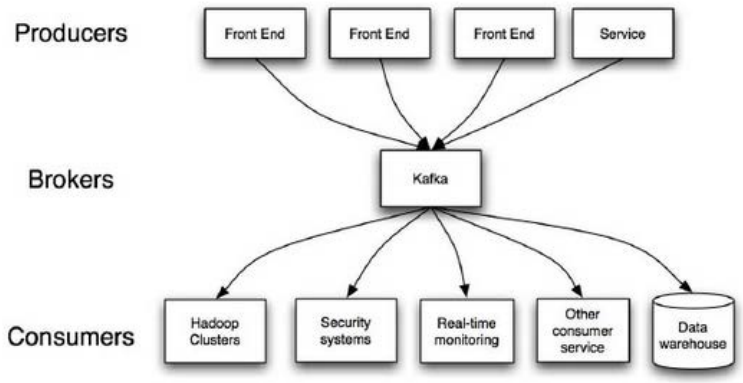


Figure 4

Kafka is explicitly distributed—producers, consumers, and brokers can all be run on a cluster of machines that co-operate as a logical group.

This happens fairly naturally for brokers and producers, but consumers require particular support. Each consumer process belongs to a consumer group and each message is delivered to exactly one process within every consumer group. Hence, a consumer group allows many processes or machines to logically act as a single consumer. The concept of consumer group is very powerful and can be used to support the semantics of either a queue or topic as found in JMS. To support queue semantics, we can put all consumers in a single consumer group, in which case each message will go to a single consumer. To support topic semantics, each consumer is put in its own consumer group, and then all consumers will receive each message. Kafka has the added benefit in the case of large data that no matter how many consumers a topic has, a message is stored only a single time.

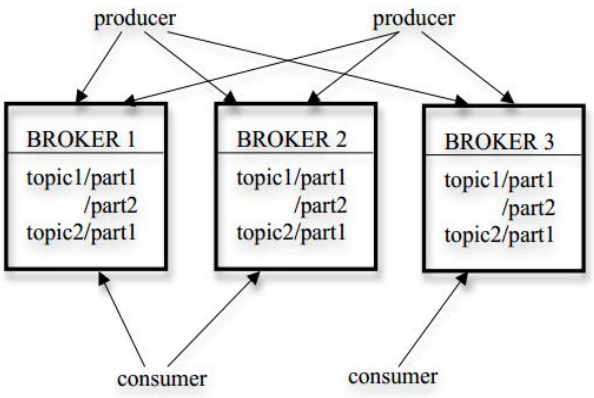


Figure 5

The overall architecture of Kafka is shown in Figure 5. Kafka is distributed in nature; a Kafka cluster typically consists of multiple brokers. To balance load, a topic is divided into multiple partitions and each broker stores one or more of those partitions. Multiple producers and consumers can publish and retrieve messages at the same time[2]

Kafka relies heavily on the file system for storing and caching messages. There is a general perception that "disks are slow" which makes people skeptical that a persistent structure can offer competitive performance. In fact, disks are both much slower and much faster than people expect depending on how they are used; a properly designed disk structure can often be as fast as the network.

The key fact about disk performance is that the throughput of hard drives has been diverging from the latency of a disk seek for the last decade. As a result, the performance of linear writes on a 6 7200rpm SATA RAID-5 array is about 300MB/sec but the performance of random writes is only about 50k/sec—differences of nearly 10000X! These linear reads and writes are the most predictable of all usage patterns, and hence the one detected and optimized best by the operating system using read-ahead and write-behind techniques.[3]

Kafka is designed around Operating System Page Cache. In computing, page cache—sometimes ambiguously called disk cache—is a "transparent" buffer of disk-backed pages kept in main memory (RAM) by the operating system for quicker access. Page cache is typically implemented in kernels with the paging memory management, and is completely transparent to applications[4]. Any modern OS will happily divert *all* free memory to disk caching with little performance penalty when the memory is reclaimed. All disk reads and writes will go through this unified cache. If your disk usage favors linear reads, then read-ahead is effectively pre-populating this cache with useful data on each disk read. This suggests a design which is very simple: rather than maintain as much as possible in-memory and flush

[2] http://research.microsoft.com/en-us/um/people/srikanth/netdb11/netdb11papers/netdb11-final12.pdf
[3] http://kafka.apache.org/design.html
[4] http://en.wikipedia.org/wiki/Page_cache

to the file system only when necessary, Kafka inverts that. All data is immediately written to a persistent log on the file system without any call to flush the data. In effect, this just means that it is transferred into the kernel's page cache where the OS can flush it later.

Kafka has a very simple storage layout. Each partition of a topic corresponds to a logical log. Physically, a log is implemented as a set of segment files of approximately the same size (e.g., 1GB). Every time a producer publishes a message to a partition, the broker simply appends the message to the last segment file. For better performance, Kafka flush the segment files to disk only after a configurable number of messages have been published or a certain amount of time has elapsed. A message is only exposed to the consumers after it is flushed.

Unlike typical messaging systems, a message stored in Kafka doesn't have an explicit message ID. Instead, each message is addressed by its logical offset in the log. This avoids the overhead of maintaining auxiliary, seek-intensive random-access index structures that map the message IDs to the actual message locations.

If the messaging system is designed around this kind of design of read ahead and write behind, how does Kafka support the Consumer State problem we defined earlier, i.e. how does Kafka keep track of which messages are being "sent" or "consumed"? Fact of the matter is, Kafka broker never keeps track of this. In Kafka, it is consumer which keeps track of the messages it consumed. Consumer is maintaining something like a watermark which tells which offset in the log segment is consumed. A consumer always consumes messages from a particular partition sequentially. If the consumer acknowledges a particular message offset, it implies that the consumer has received all messages prior to that offset in the partition. This provides flexibility to the consumer to consume older message by lowering the watermark. Typically, Consumer stores the state information in Zookeeper which is used for Distributed consensus service. Otherwise, Consumer can maintain this watermark level in any data structure it wishes, which depends on the Consumer. For instance, if Hadoop is consuming messages from Kafka, it can store the watermark value into HDFS.

Along with the architectural details mentioned above, Kafka also has many advanced configurations such as Topic Partitioning, Automatic Load Balancing, and so on. More advanced details can be found on the Kafka website.

This design of Kafka makes it highly scalable, able to process millions of messages per second. The producer of real time data can write messages into Kafka cluster, and the real time consumer can read the messages from Kafka cluster for parallel processing.

Apache Storm is one such distributed real time parallel processing platform developed by Twitter. With Storm and Kafka, one can conduct stream processing at linear scale, assured that every message is reliably processed in real-time. Storm and Kafka can handle data velocities of tens of thousands of messages every second.

Let us now look at the Storm architecture and how it can connect to Kafka for real time processing.

Real Time Data Processing Platform: Storm

Storm is a free and open source distributed real time computation system. Storm has many use cases: real time analytics, online machine learning, continuous computation, distributed RPC, ETL, and more. Storm is fast; a benchmark clocked it at over a million tuples processed per second per node. It is scalable, fault-tolerant, guarantees your data will be processed, and is easy to set up and operate.[5]

The concept behind Storm is somewhat similar to Hadoop. In Hadoop cluster, you run Map Reduce Job; in Storm Cluster, there is Topologies. The core abstraction in Storm is the "stream". A stream is an unbounded sequence of tuples as shown in Figure 6.

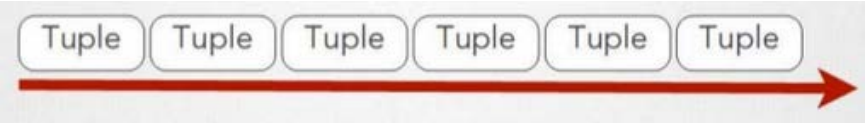
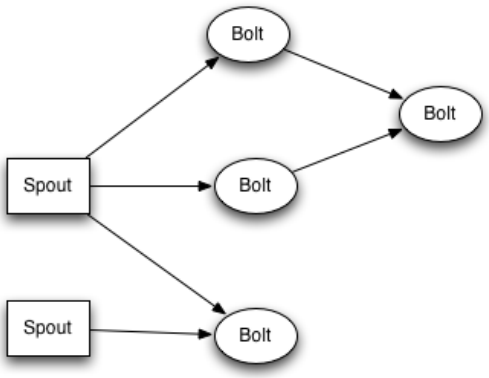


Figure 6

Storm provides the primitives for transforming one stream into a new stream in a distributed and reliable way. For example, you may transform a stream of tweets into a stream of trending topics.

[5] http://storm-project.net/



Storm Topologies are combination of Spouts and Bolts. Spouts are where the data stream is injected into the topology. Bolts process the streams that are piped into it. Bolt can feed data from spouts or other bolts. Storm takes care of parallel processing of spouts and bolts and moving data around.

Figure 7

Figure 8 shows an example of Spout, which emits tuple stream.

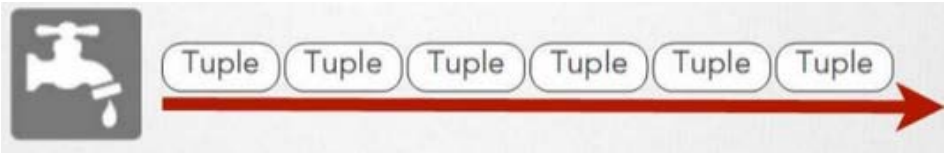


Figure 8

Figure 9 depicts Bolts, which process tuple and emits new stream.

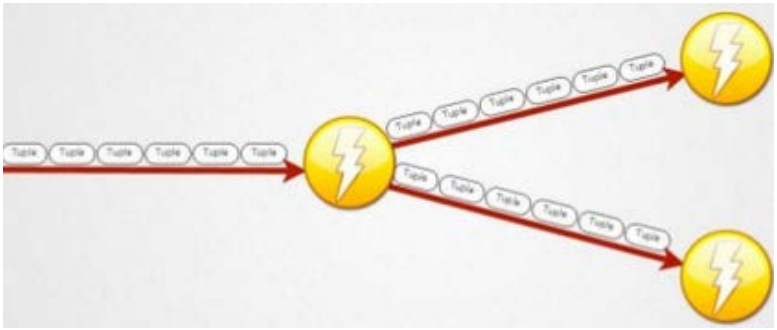


Figure 9

Each node in storm topologies can execute in parallel. One can specify how much parallelism is required for each node. Storm will spawn that many threads to process them across a cluster.

Storm has three high level entities which actually run Topologies in Storm cluster.

1. Worker Process
2. Executors
3. Task

Here is the simple illustration of their relationship.[6]

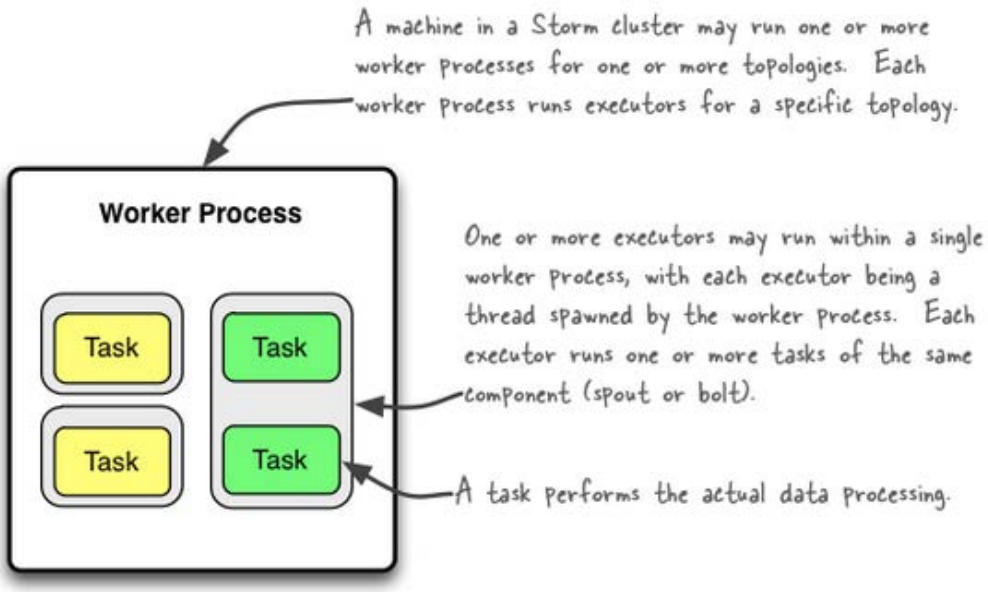


Figure 10

The Worker Process executes the subset of the topology. Worker Process runs one or more executors from one or more components (Spouts or Bolts). In a given topology, there could be many such worker processes running across machines.

The executor is the thread spawned by Worker Process and it will run a task. The task performs the actual processing, i.e. each spouts and bolts in the cluster.

Figure 10 shows a simple Storm topology with One Spout and Two Bolts.[7]

[6] http://www.michael-noll.com/blog/2012/10/16/understanding-the-parallelism-of-a-storm-topology/#what-is-storm

[7] https://github.com/nathanmarz/storm/wiki/Understanding-the-parallelism-of-a-Storm-topology

Configuring the parallelism of a simple Storm topology

```
1  Config conf = new Config();
2  conf.setNumWorkers(2); // use two worker processes
3
4  topologyBuilder.setSpout("blue-spout", new BlueSpout(), 2); // parallelism hint
5
6  topologyBuilder.setBolt("green-bolt", new GreenBolt(), 2)
7                 .setNumTasks(4)
8                 .shuffleGrouping("blue-spout");
9
10 topologyBuilder.setBolt("yellow-bolt", new YellowBolt(), 6)
11                .shuffleGrouping("green-bolt");
12
13 StormSubmitter.submitTopology(
14         "mytopology",
15         conf,
16         topologyBuilder.createTopology()
17     );
```

Figure 10

At line number 2, we configure the number of Worker Process as two.

Here, the Blue Spout is the origin of the tuple stream and has the parallelism hint specified as two. The Blue Spout is connected to Green Bolt and the parallelism hint is again two for this bolt but the number of task specified is four. The green bolt is connected to Yellow Bolt, for which the parallelism hint is six. For this topology, Figure 11 is the pictorial depiction of the total number of worker, executor, and task in the topology.

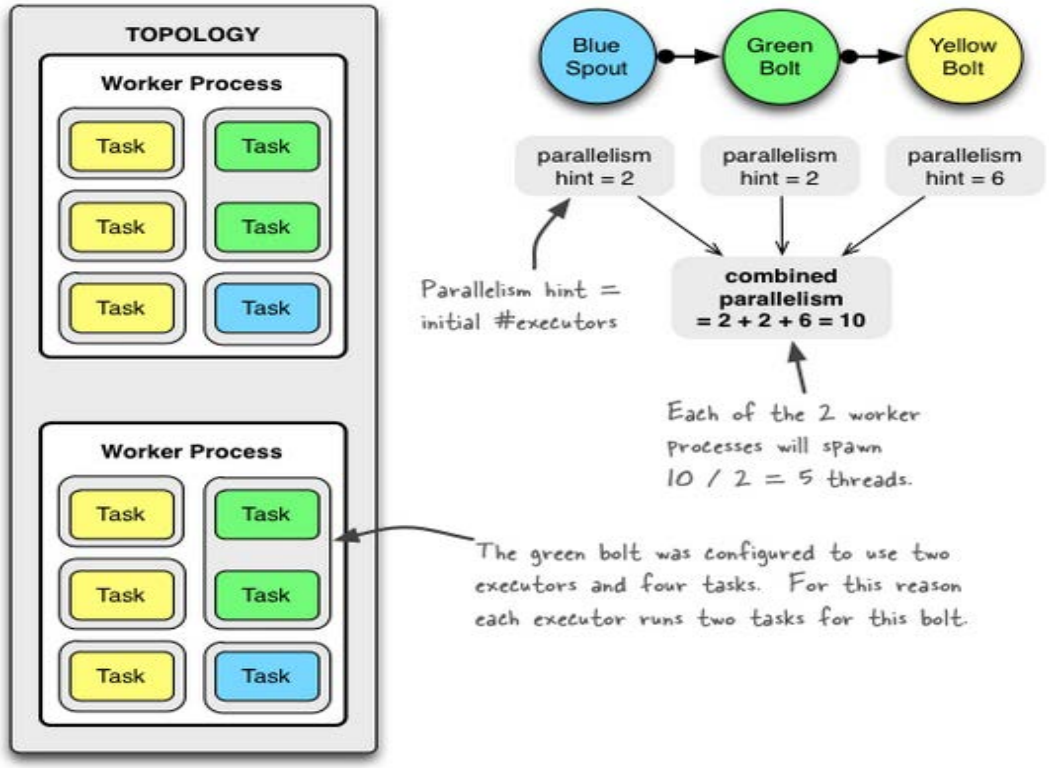


Figure 11

The fundamental concept is if we need to process stream of tuples in parallel in Storm, we need to define our own Spouts and Bolts, and need to configure Storm to optimizing its parallelism across nodes. While there are many advanced concepts—such as how to group tuple and give it to Bolts—we are not going into such details in this article. However, the Storm wiki has nice documentation which explains those concepts.

As we learned earlier, Apache Kafka is one of the highly scalable distributed messaging system, and if Storm need to fetch the messages from Kafka Cluster and process them in parallel, a Storm-Kafka spout is needed. This spout should be able to read the message stream from Kafka, and emit that stream to Storm Bolts (which we will define later) for downstream processing. A major challenge in Storm-Kafka spout is to keep track of the message offset (till offset messages are consumed by Storm from Kafka). As explained earlier, this is most important for implementing any connectors from Kafka as the Kafka message consumer needs to keep track of the message offset.

There is already an open source Storm-Kafka spout available in the Gut Hub which we used for our implementations. The code is available here: https://github.com/nathanmarz/storm-contrib/tree/master/storm-kafka .

Once we use the Storm-Kafka spout and configure the spout to read messages from a specific Kafka Topic, Storm will start fetching the data. At this point we need to configure other bolts in Storm topology to work on the tuple stream fetched from Kafka.

We will discuss what other Bolts we need to use, but before that, let's look at a case study where we will try to predict a problem beforehand using the Real Time processing architecture. Here we need to detect a failure in a given network system based on the events emitted from various sensors. Once we understand the case study, we will come back to the Real Time system.

Case Study: Networking Fault Prediction

Network topology size is increasing day by day. Customers have thousands of routers, switches, servers, chassis, cards, VLANs, cables, ports, and interfaces in their topology. Some products are able to determine the root cause of a problem within such complex topology. However, customers would prefer to be notified before the actual problem occurs. The idea is to predict the problem before it appears by monitoring real time events.

Network management products monitor ports, cards, switches, and so on through SNMP/syslog and generate producing event logs indicating the status of each of these as UP or DOWN.

By collecting those logs for a specific topology for a certain amount of time (for example, 6 months), and analyzing the sequence of event patterns, a model to predict a fault (with probability) can be built.

Let's take a simple use case.

We have a topology which consists of 4 switches, which are connected to each other via cables. Each switch has 2 cards, each card has 2 ports, and each port is connecting to 2 cables (Links).

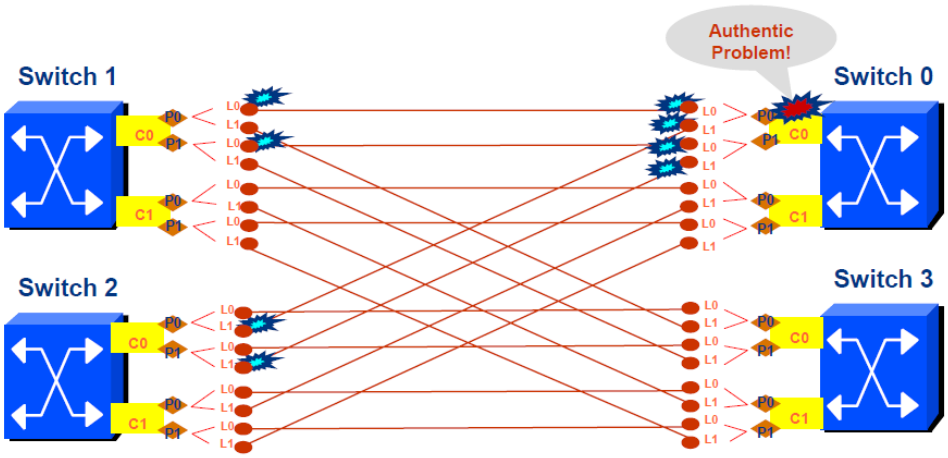


Figure 12

In the past, we have learned from the logs that, when Switch0 goes down, the following sequence of events occur before that:

(Here S0C0P0L0... mean Link 0 of Port 0 of Card 0 of Switch 0..)

(S0C0 mean Card 0 of Switch 0, etc.)

1. S0C0 is down
2. S0C0P0 is down
3. S0C0P1 is down
4. S0C0P0L0-S1C0P0L0 link is down
5. S0C0P0L1-S2C0P0L1 link is down
6. S0C0P1L0-S1C0P1L0 link is down
7. S0C0P1L1-S2C0P1L1 link is down
8. S0C1 is down
9. S0C1P0 is down
10. S0C1P1 is down
11. S0C1P0L0-S1C1P0L0 link is down
12. S0C1P0L1-S2C1P0L1 link is down
13. S0C1P1L0-S1C1P1L0 link is down
14. S0C1P1L1-S2C1P1L1 link is down

Here you can see that the first set of events (1-7) occur when Card0 of Switch0 goes down. The next set of events (8-14) occurs when Card1 of Switch0 goes down. For Switch0 to be down,

both Card0 and Card1 need to be down. Also, from past event logs we found that if Card0 of Switch0 goes down, there is a 50% possibility that Switch0 will also go down. With this information, if the Real Time stream processing system identify events 1-7 (events related to Card0 of Switch0 down), it can predict with 50% probability that Switch0 will go down in the future without waiting for the next set of events (8-14) to occur.

This is an example of predicting the problem beforehand.

As you can see, there could be numerous such predictions possible in a network topology given it can have hundreds of switches, routers, cards, links, and so on. There could be different sequence of events for different types of failures. The ability to build models around most major failures in the system to predict a failure by monitoring the real time event streams will be an enormous benefit for business to take preventive action immediately to avert catastrophic network failure.

However, doing this kind of prediction is not easy task. The whole prediction system should have two phases; offline and online. The offline phase collects, filters, and builds a statistical model for sequence prediction, and then uses this model in online / real time system to perform actual prediction. We will examine the details of offline and online systems in the next few sections.

The Offline Process: Model Building

The failure prediction system is one of the major use cases in real time analytics. As mentioned earlier, to detect failure in a given stream of sensor data, we need to first define normal behavior. For this we need to build model around the historical sensor data. To build the model, we can use an offline Batch Processing system such as Hadoop to extract, transform, and load historical sensor records from logs. For offline data processing, we used Hadoop to pull the historical data stored in the Kafka cluster and then process the same in Hadoop to partition the log records and then try to extract a correct sequence of events for given network segment. Once this model is built, it is used in the real time system (Storm/Kafka) for fault prediction. A representative depiction of the architecture is shown in Figure 13.

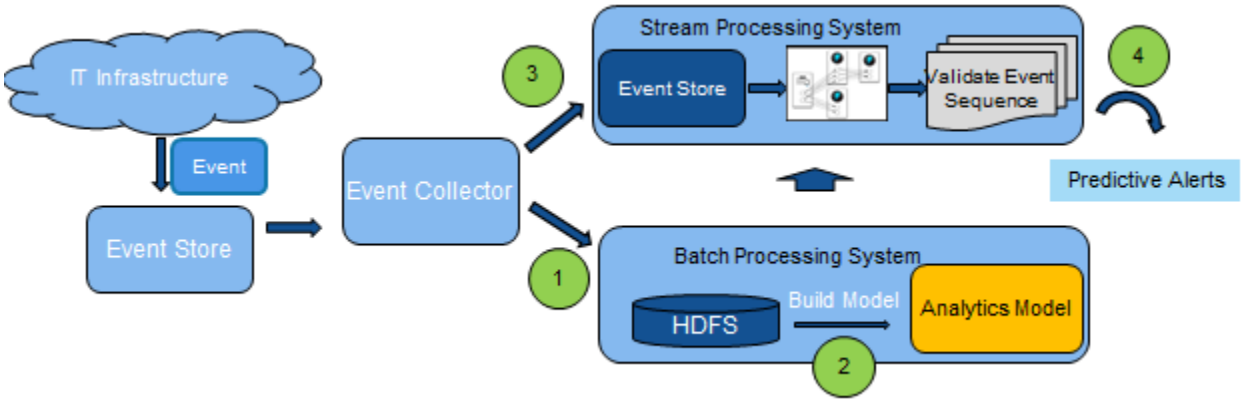


Figure 13

The Batch Processing System (1 and 2) is the offline process of Model building, and Stream processing System (3 and 4) is the online process for Real Time prediction.

In this section, we will explain the offline process of Model Building using the Hadoop machine learning library, Mahout. As you already know, all sensor data is being collected at the Kafka cluster. For writing data to the Kafka cluster, the Sensor event generator code needs to be modified to act as a Kafka Producer. If that is not possible, we may need to develop wrapper Kafka Producer to read the sensor data and write to Kafka cluster. For this case study, we used a simulator to generate the network events and writing them into Kafka.

For offline process, Hadoop is used as the Kafka Consumer. To read data from Kafka, Hadoop needs to access Kafka to read the segment files stored in Kafka cluster. Although the details of Hadoop internals is beyond the scope of this article, it bears mentioning that Hadoop can connect to an external data source such as Kafka provided the Map Reduce layer can use the custom Record Reader and Input Format to read and split the records in sizable chunks for parallel processing in the Map Reduce engine. One such open source implementation of HDFS Kafka Consumer is available in github which we used for case study. The code is available at:

https://github.com/kafka-dev/kafka/tree/master/contrib/hadoop-consumer

Here using this custom Kafka consumer, Hadoop can consume data from Kafka for a given Topic. We previously discussed that any Kafka Consumer needs to track the message offset up to the point the message has been consumed. In a similar way, the Hadoop Kafka consumer also keeps track of consumed offset and maintain it in HDFS.

The Hadoop Map Reduce job is written to filter the specific sequence of events for a given Topic, and use Mahout Library to build the Hidden Markov Model for fault detection. We used Hadoop version 0.20.2. and Mahout version 0.7.

We will now discuss the Hidden Markov Model; what it is and how it can be used for fault detection.

Hidden Markov Model

Hidden Markov Models are used in multiple areas of Machine Learning, such as speech recognition, handwritten letter recognition, natural language processing, and so on. Reviewing a few examples will make it easier to grasp the concept.[8] We are not going to discuss the mathematics part of the model, as it will be very complex to discuss in this article.

Consider two friends, Alice and Bob, who live far apart from each other and who talk together daily over the telephone about what they did that day. Bob is only interested in three activities: walking in the park, shopping, and cleaning his apartment. The choice of what to do is determined exclusively by the weather on a given day. Alice has no definite information about the weather where Bob lives, but she knows general trends. Based on what Bob tells her he did each day, Alice tries to guess what the weather must have been like.

Alice believes that the weather operates as a discrete Markov chain. There are two states, "Rainy" and "Sunny", but she cannot observe them directly, that is, they are hidden from her. On each day, there is a certain chance that Bob will perform one of the following activities, depending on the weather: "walk", "shop", or "clean". Since Bob tells Alice about his activities, those are the observations. The entire system is that of a hidden Markov model (HMM).

Alice knows the general weather trends in the area and what Bob likes to do, on average. In other words, the parameters of the HMM are known. They can be represented as:

[8] http://en.wikipedia.org/wiki/Hidden_Markov_model

```
states = ('Rainy', 'Sunny')

observations = ('walk', 'shop', 'clean')

start_probability = {'Rainy': 0.6, 'Sunny': 0.4}

transition_probability = {
   'Rainy' : {'Rainy': 0.7, 'Sunny': 0.3},
   'Sunny' : {'Rainy': 0.4, 'Sunny': 0.6},
   }

emission_probability = {
   'Rainy' : {'walk': 0.1, 'shop': 0.4, 'clean': 0.5},
   'Sunny' : {'walk': 0.6, 'shop': 0.3, 'clean': 0.1},
   }
```

Figure 14

In this piece of code (Figure 14), start probability represents Alice's belief about which state the HMM is in when Bob first calls her (all she knows is that it tends to be rainy on average). The transition_probability represents the change of the weather in the underlying Markov chain. In this example, there is only a 30% chance that tomorrow will be sunny if today is rainy. The emission_probability represents how likely Bob is to perform a certain activity on each day. If it is rainy, there is a 50% chance that he is cleaning his apartment; if it is sunny, there is a 60% chance that he is outside for a walk.

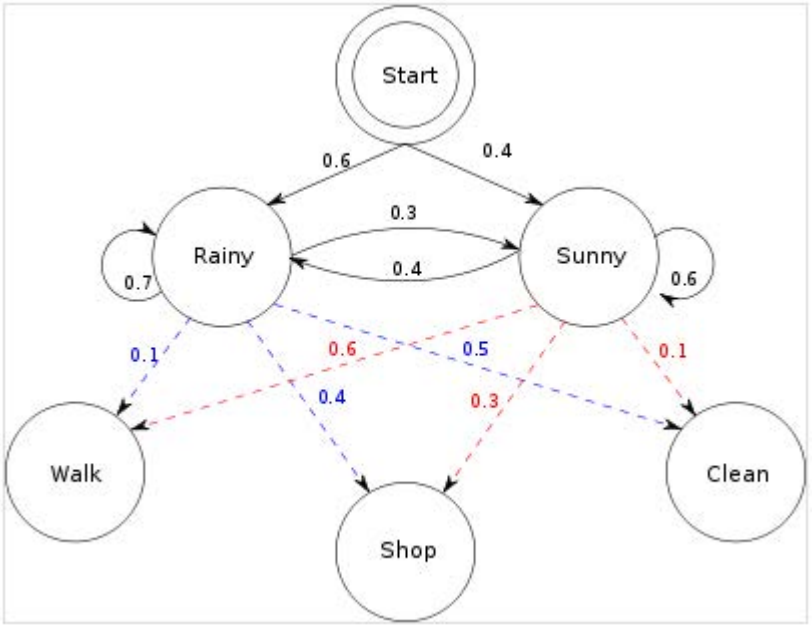


Figure 15

So, as shown in Figure 16[9], the Hidden Markov Model consist of a Transition Probability Matrix T (Tij is the probability of transition from Hidden state Ti to Tj), the Observer or Emission matrix O (Oij is the probability of transition from Oi to Oj), and Vector PI as the start probability.

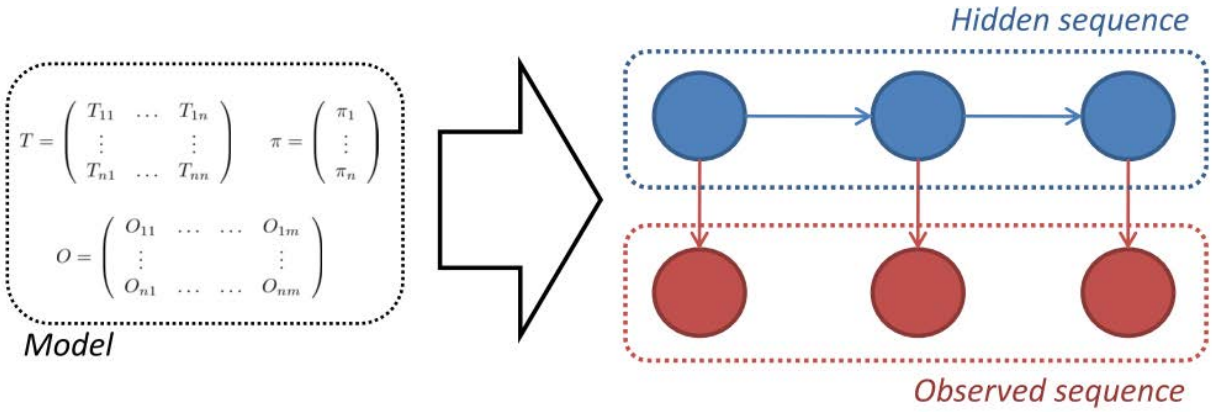


Figure 16

Some fields where HMM is used:

- Cryptanalysis
- Speech recognition
- Speech synthesis
- Part-of-speech tagging
- Machine translation
- Gene prediction
- Alignment of bio-sequences
- Activity recognition
- Protein folding

The HMM problem domain is mainly categorized into Evaluation, Decoding, Learning (Supervised and Unsupervised).

Problem	Given	Wanted
Evaluation	*Model, Observed Sequence*	Likelihood the model produced the observed sequence
Decoding	*Model, Observed Sequence*	Most likely hidden sequence
Learning (unsupervised)	*Observed Sequence*	Most likely model that produced the observed sequence
Learning (supervised)	*Observed- & Hidden sequence*	Most likely model that produced the observed & hidden sequence.

Our case study mainly comes under the Unsupervised learning problem where we only have the Observed sequence (Sequence of events emitted by network sensors) and we need to build the

[9] http://isabel-drost.de/hadoop/slides/HMM.pdf

most probable Model which is best suited for this observed sequence. There are algorithms available for building the Model from observed sequence called Baum-Welch algorithm.[10]

Once the Hidden Markov Model is built, we do the Evaluation where we perform the Likelihood test to check the probability for a given a sequence, generated by the Model. The following small code in Mahout will help to understand this.

```
int[] observedSeq = {0,0,1,1,2,1,0,0,1,1,2,1};
HashMap<String, Integer> map = new HashMap<String,Integer>();
map.put("A", 0);
map.put("B", 1);
map.put("C", 2);

HmmModel model = new HmmModel(2,3);
model = HmmTrainer.trainBaumWelch(model,observedSeq,1.0,5,true);
model.registerOutputStateNames(map);

int[] givenSeq = {2,2};
double steps = HmmEvaluator.modelLikelihood(model, givenSeq, true);

System.out.println(steps);
```

Here we are building one HmmModel using the observed sequence of three states, A (0), B (1) and C (2).

We assumed that the HmmModel has 2 hidden states and 3 observed state.

We used Mahout HmmTrainer to train using Baum-Welch algorithm. At this point, the model is built and is ready to use to validate any given sequence.

Assume that a given sequence is {2, 2}. Once we test for the likelihood of this sequence, we get the probability of around 2%. But if we test with given sequence {0, 1}, the likelihood probability will be around 19%. So you can see, if the model is correct, it can predict if a given sequence is faulty sequence or not, i.e. the likelihood factor of a given sequence generated by Model.

Similarly, when we build the HmmModel using Hadoop by collecting the Topic data (generated by network sensors) from Kafka, and if at real time, the probability of a given sequence of an event's likelihood probability deviates from a certain threshold, we can predict that the given sequence is a faulty sequence which is not part of the model and the faulty sequence is there in stream data because of system issues.

[10] http://en.wikipedia.org/wiki/Baum%E2%80%93Welch_algorithm

With this knowledge, let us move to the Real Time processing part again where the Storm processing engine retrieved the stream events from Kafka and tries to calculate the likelihood probability of the collected sequence of data against the HmmModel built using Hadoop. Let us see how this can be done in Storm.

The Online Process: Model Validation

For our case study we used following version of Storm and Kafka

Storm Version: 0.8.0

Kafka Version: 0.7.0

Storm-Kafka Spout: 0.7.2

Setting up the development environment will not be discussed here. However, you can find those details in Storm-Starter project in github (https://github.com/nathanmarz/storm-starter).

The code snippet to connect to Kafka cluster from Storm using Spout Configuration object is shown below.

```
List<String> host = new ArrayList();
host.add("localhost");
SpoutConfig spoutConfig = new SpoutConfig(host, 1, "HMM", "/foo", "foo");
spoutConfig.zkServers = ImmutableList.of("127.0.0.1");
spoutConfig.zkPort = 2181;
spoutConfig.scheme = new StringScheme();
spoutConfig.zkRoot ="/brokers";
spoutConfig.forceStartOffsetTime(-1);

KafkaSpout spout = new KafkaSpout(spoutConfig);
```

Here we are creating a Kafka spout configuration by specifying the Topic name (“hmm”), the Zookeeper server details which Kafka used for managing Brokers, the path in Zookeeper (/foo) where the Kafka spout will store the consumed offset details, and so forth.

The forceStartOffset is set to -1, meaning this Spout will read the new messages from the last read offset. As discussed earlier, the Kafka Spout can read already consumed messages also. But for our case study, we are fetching only the fresh messages.

The challenge now is, once we configure this Kafka Spout, Storm will start reading the real time event stream from Kafka as and when network sensors produce the events and write to Kafka.

The Kafka spout will flood the Storm topology with the tuple stream. We should now be able to group or batch these tuple streams and apply the model.

Event processing is a method of tracking and analyzing (processing) streams of information about things that happen, and deriving a conclusion from them. Complex event processing (CEP), is event processing that combines data from multiple sources to infer events or patterns that suggest more insights. The goal of CEP is to identify meaningful events (such as opportunities or threats) and respond to them as quickly as possible.[11]

One such open source CEP engine is Esper (http://esper.codehaus.org/).

Esper offers an Event Processing Language (EPL) which is a declarative language for dealing with high frequency time-based event data.

Examples of applications using Esper are:[12]

- Business process management and automation (process monitoring, BAM, reporting exceptions, operational intelligence)
- Finance (algorithmic trading, fraud detection, risk management)
- Network and application monitoring (intrusion detection, SLA monitoring)
- Sensor network applications (RFID reading, scheduling and control of fabrication lines, air traffic)

One of the interesting things Esper can do is event batching, i.e. combining events for, say, 1 minute batch. In this case, Esper can process all tuple streams coming in a given batch together. This is very important for detecting failure. For instance, if events can be batched for the previous 1 minute and a fault can be found within this batch ("given sequence" in Hidden Markov) it can be predicted immediately. For this case study, we did event batching every 30 seconds and, in our case, sensors simulators generate data very fast so we needed to keep batch size small to enable fast predictions. For a real life problem, the batch size needs to be set diligently.

The Esper CEP maintains a batch buffer to keep all the events coming into the Esper. We have first-hand experience with this as we have used this buffer for applying the model and calculating the likelihood probability.

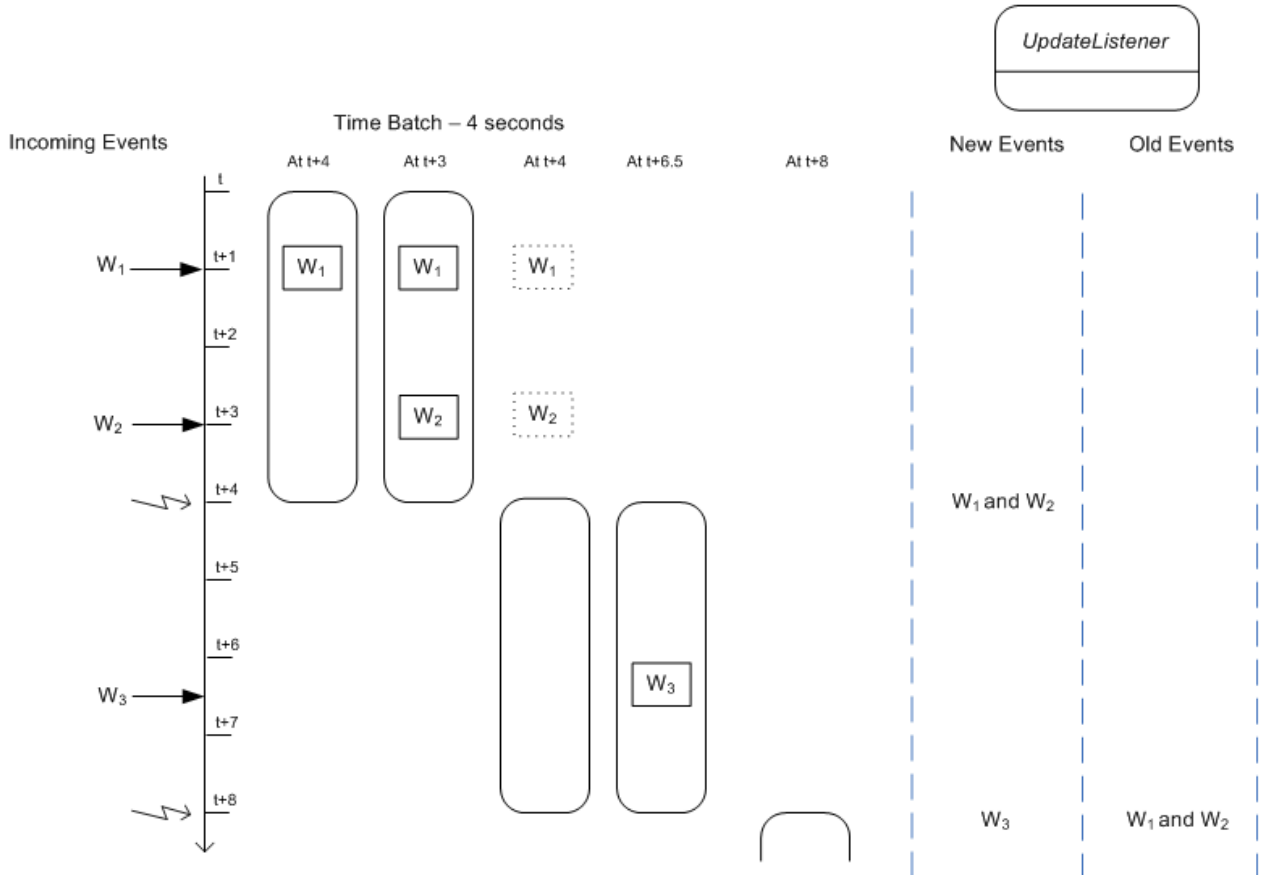
[11] http://en.wikipedia.org/wiki/Complex_event_processing
[12] http://esper.codehaus.org/

Assume that we want to find the latest withdrawals from a given bank in a four-second time batch window. Below is the query executed in Esper.

```
select * from Withdrawal.win:time_batch(4 sec)
```

The time batch view buffers events and *releases them every specified time interval in one update*. Time windows control the evaluation of events, as does the length batch window.[13]

To explain this, the diagram below depicts starting at a given time **t** and displays the contents of the time window at **t + 4, t + 5** seconds, and so on.



The activity as illustrated by the diagram: (The W stands for withdrawals)

- At time t + 1 seconds, event W1 arrives and enters the batch. No call to inform update listeners occurs.

[13] http://esper.codehaus.org/esper-1.7.0/doc/reference/en/html/outputmodel.html

- At time t + 3 seconds, event W2 arrives and enters the batch. No call to inform update listeners occurs.
- At time t + 4 seconds, the engine processes the batched events and starts a new batch. The engine reports events W1 and W2 to update listeners.
- At time t + 6.5 seconds, event W3 arrives and enters the batch. No call to inform update listeners occurs.
- At time t + 8 seconds, the engine processes the batched events and starts a new batch. The engine reports event W3 as new data to update listeners. The engine reports the events W1 and W2 as old data (prior batch) to update listeners

Esper maintains a New Event Buffer and Old Event buffer inside its Update Listener. Hence, at any given point of time, the latest events for a given batch can be obtained.

With this concept in mind, let us now see how Storm can handle the Batch events for Esper to process. As we learned earlier, in Storm, all we need is Spouts and Bolts. We already defined our Kafka Spout to read stream from Kafka. Now we have to prepare our Esper Bolt to perform event processing in Esper. There is already an open source Esper bolt available in github (https://github.com/tomdz/storm-esper) which we used in our case study.

A code snippet of how the Esper Bolt is configured to consume the tuple stream from Kafka Spout is shown below.

```
TopologyBuilder builder = new TopologyBuilder();
List<String> host = new ArrayList();
host.add("localhost");
SpoutConfig spoutConfig = new SpoutConfig(host, 1, "HMM", "/foo", "foo");
spoutConfig.zkServers = ImmutableList.of("127.0.0.1");
spoutConfig.zkPort = 2181;
spoutConfig.scheme = new StringScheme();
spoutConfig.zkRoot ="/brokers";
spoutConfig.forceStartOffsetTime(-1);

KafkaSpout spout = new KafkaSpout(spoutConfig);

EsperBolt esperBolt =
        new EsperBolt.Builder(model)
                    .inputs()
                       .aliasComponent("spout")
                       .withFields("str")
                       .ofType(String.class)
                       .toEventType("HMMState")
                    .outputs()
                       .outputs().onDefaultStream().emit("state")
                    .statements()
                       .add("select str as state from HMMState.win:time_batch(30 sec)")
                    .build();

builder.setSpout("spout", spout);
builder.setBolt("bolt1", esperBolt).shuffleGrouping("spout");

Config conf = new Config();
conf.setDebug(true);
LocalCluster cluster = new LocalCluster();
cluster.submitTopology("test", conf, builder.createTopology());
```

Here, the complete Storm Topology code is shown where Esper Bolt and Kafka Spouts are configured together using the Topology Builder. The interesting part is the section highlighted above. This is the Esper Processing Language (EPL) which can take the tuple stream from Spout and batch it in every 30 seconds. If we run this Storm topology, and if events are pumped into Kafka cluster (in any random frequency), we can see that Storm emits those events every 30 seconds i.e. Storm will emit all events together which occur in a 30 seconds batch.

At this point, we have a good handle on the event stream and can do further processing on this batch event. As already discussed, we need to apply the HmmModel into this batch event continuously and, finally, Storm should emit the likelihood probabability. To apply the HmmModel, we basically modified the open source EsperBolt and used the event buffer described earlier. Here are the details of changes done in the open source code of EsperBolt.

First, we modified the Bolt declaration and passed the instance of HmmModel object. The EsperBolt open source implementation does not take any parameter.

```
EsperBolt esperBolt =
        new EsperBolt.Builder(model)
                    .inputs()
                       .aliasComponent("spout")
                       .withFields("str")
                       .ofType(String.class)
                       .toEventType("HMMState")
                    .outputs()
                       .outputs().onDefaultStream().emit("state")
                    .statements()
                       .add("select str as state from HMMState.win:time_batch(5 sec)")
                    .build();
```

Below is the corresponding change in the Builder method of EsperBolt to take the HmmModel.

```
public class EsperBolt extends BaseRichBolt implements UpdateListener
{
    private static final long serialVersionUID = 1L;
    private static HmmModel model;

    public static class Builder
    {
        protected final EsperBolt bolt;

        public Builder(HmmModel model)
        {
            this(new EsperBolt());
            EsperBolt.model = model;
        }
```

At this point, the bolt has access to the model to apply on the stream. As described earlier, the Batch events accumulated into a buffer within the Update Listener method. Below is the same method in EsperBolt class which is modified to get hold of this buffer.

```
public void update(EventBean[] newEvents, EventBean[] oldEvents)
{
    int[] array = new int[newEvents.length];
    if (newEvents != null) {
        EventTypeDescriptor eventType = null;
        int pos = 0;
        for (EventBean newEvent : newEvents) {
            eventType = getEventType(newEvent.getEventType().getName());

            if (eventType == null) {
                // anonymous event ?
                eventType = getEventType(null);
            }
            if (eventType != null) {
                String state = (String)newEvent.get(eventType.getFields().get(0));
                if(state != null || !state.equalsIgnoreCase(""))
                    array[pos] = Integer.parseInt(state);
            }
        }

        double steps = HmmEvaluator.modelLikelihood(model, array, true);
        collector.emit(eventType.getStreamId(), Utils.tuple(Double.toString(steps)));
    }
}
```

Here, the update method is called every time a new time batch window executed (i.e. called every 30 seconds in our case). We applied the likelihood test on the batched stream and emit the likelihood count. The likelihood test is done against the model which passed to Bolt during its initialization.

With the above changes, if we run the Topology, we can see that Storm will emit a Probability number every 30 seconds as calculated above. We can further connect this EsperBolt to some other Bolt which can store this number into a Real Time visualization framework. We are not showing any visualization aspect of this prediction in this article.

A missing piece is; how does the Storm will get hold of the Model which was built offline using Hadoop? This can be done a couple of ways. First, the HmmModel built in Hadoop can be serialized into disk, and the Storm Topology can de-serialize it and use it in the Esper Bolt. Another option; Hadoop can load the HmmModel object into some Caching layer, and Storm Topology can get it from the Cache. Ideally, the Cache solution will work better, as we can store multiple Models built for multiple predictions, and each prediction can be associated with a given topic in Kafka. With this concept, we can have individual Kafka spout (for each Topic), and corresponding Esper Bolt (Initialized with the Model fetched from Cache for that Topic) in a given Storm topology.

Case Study: Solution

For the case study, we have created simulators for different types of sensors for routers, switches, VMs, storage, and so forth. Different sensors can emit different kind of events related to their status and associated devices. In our case study, we are emitting events related to switches and associated devices such as card, port, link, and so on. We have categorized each sensor type to a topic (for example, a topic related to switch or router) in Kafka, i.e. one Topic for router and its associated device-related sensors, one Topic for switch and its associated device related sensors, and so on. The offline model building is done by using different Map Reduce jobs specific to a topic. Which mean one Hadoop job is written to fetch one topic stream from Kafka and then build the Model around it.

As mentioned earlier in our case study, we wanted to predict the failure probability of a device from a given sequence of events. Thus, we make the model from a failure sequence of any historical failures for a device, i.e. for VM-related events, if a fan goes down, the CPU will heat up, causing CPU utilization to decrease, which causes slow response time, and finally, the VM will be down. This whole sequence will be my failure sequence. We numbered the sequence so that the same can be used in Mahout to build the HmmModel. Using Hadoop Map Reduce, we have to trace these types of sequences from the Logs and then build the HmmModel around it.

Once the model is built and used in Storm (as described previously), the prediction criteria will be a a little different. This time we have to find the likelihood of a given sequence, which has high likelihood probability. As we are predicting failure, and model is built around a failure sequence, real time sequence probability with higher likelihood values mean the given sequence is moving toward failure. Knowing this, we can predict failure before it happens.

Summary

The scope of the work in this article is limited to predicting individual sequence of events for a topic. But in real world scenarios, failures in one type of topic can cause failure to others. For example, switch failure associated with a router can cause the router to fail, which may cause VMs associated with those routers to become non-responsive, causing applications running on the VM to fail, and so on. Thus, correlating failure events between topics is one of the major use cases for this type of distributed network. We have proposed an architecture using Storm and Kafka, which is a very highly scalable distributed real time processing platform; it can handle and scale to a very high input stream rate. Also, using Esper will enable us to perform event

correlation across topics. We have not evaluated cross-topic event correlation, but it is possible with the given architecture.

One more advanced use case for network failure prediction is the way the Hidden Markov Model can be built. Here we have built the Model for a given device (i.e. a switch). But there could be hundreds of switches in a network and there will be cases where the failure sequence of one set of switches will differ from another set of switches based on different configurations or topologies. The model needs to factor this in since one HmmModel for a given set of switches cannot be used for another set of switches. We can either create different Topic streams for different categories (i.e. model for switch1 and model for switch2), or we can use the Combined Hidden Markov Model[14], if possible.

Another important aspect of Real Time prediction is continuous model updating. As the sequence of events might change because of new configurations, changes in infrastructure, and so forth, the Model should be able to evolve over time to keep up with the environmental change. This will make the prediction process accurate.

Nevertheless, real time fault prediction is a difficult problem to solve and it is much more difficult when it comes to Big FAST Data. In this article, we proposed an architecture which can enable engineers and scientist to use and build a solution around it. With the advent of technology revolution around Big Data Analytics, we can definitely attempt to solve these types of complex problem on Big Fast Data at real time.

[14] http://www.stats.ox.ac.uk/__data/assets/file/0016/3328/combinedHMMartifact.pdf