Cloud Based Real Time Streaming Systems and Architectures

Guru Pradeep Reddy, 14C0246
gurupradeept@gmail.com, 8123289195

---

1. INTRODUCTION TO STREAMING

Streaming real time processing have become very common as of few years ago, and there is lot of research going on these systems. The demand for the real time processing increased because it is one of the main requirement of modern business. Big data and cloud computing have changed the face of modern IT services and infrastructure and processing of the big data has become mandatory for businesses to survive. The nature of data is evolving and need batch processing to process them. So there is definitely place for Hadoop and other batch processing systems. But recently there is a different bread of data, real time streaming data such as real time search where a company would have millions of requests to address each hour or something like high frequency trading where the machines have to decide in milliseconds, nanoseconds whether they should buy order or sell order on a certain stock or for example again in the case of social network, as a user you wouldn't want the provider to only have a batch processing system. Every hour we update the status of everyone and within the hour you just keep reading what was there about an hour ago. That's not what you want. You want to see if some of your friends post something on Facebook. You want to see it within a few seconds. So, Real time streaming is a system in which we process large volumes of the data is processed is very quickly and information obtained can be used to react the changing conditions in real time. These Large quantities of processed stream data enable the organizations to react to any potential threats or fraudulent activity. This is opposite to what we do in traditional data base systems where we first store the data and later we query the data, but here we get the information from the data while it is in motion thereby meeting the requirements of real time systems.

So, any system that has a stream of events that's flowing through this system for a given rate of data, data rate, we consider it sort of a real-time streaming system. If it cannot keep up with the event rate, say suddenly there's just so much demand. And the number of machines that you have set aside for processing this streaming data set is not enough. We want the system to degrade gracefully, for example, by eliminating events. We don't want the whole to go belly up and crash. We want the system to keep continuing, maybe drop some of the events.

As such mentioned before, batch processing systems like Hadoop and MapReduce. They store and process data in batches, at scale, which is good for some use cases, but it's not useful for real-time systems. So, to meet the requirements so many tools have come in recent years and some of the popular ones are Apache Flink, Apache Storm, Spark Streaming and there are various types of architectures like Lambda architecture and Kappa architecture to meet the needs of real time processing systems. Apache storm and Spark streaming are explained in greater detail in below sections and also Lambda and Kappa architectures are also explained below. Actually there was a set of older, non-cloud, non-big data systems that never the less we're trying to address real-time streaming. IBM System S is a prominent commercial example. Still very readily used in the industry.

2. APACHE STORM

This section explains in detail about Apache storm one of the popular system used to solve the problem of the real time processing which is crucial to many modern businesses. Apache storm is distributed real time processing system which is fault tolerant and horizontally scalable. It is a stateless and uses Apache Zookeeper to manage all the systems in the distributed cluster. It is simple to work with and can solve many large scale problems.

Apache storm is Open source, robust and user friendly. It is fault tolerant, reliable, scalable and it incredibly fast and allows real time data processing. In the next section some of the key concepts of Apache storm.

2.1 Core Concepts

Storm usually reads the streaming data from a source and does some processing of the data and returns the output to some sink. The following diagram depicts the key concepts present in the storm.

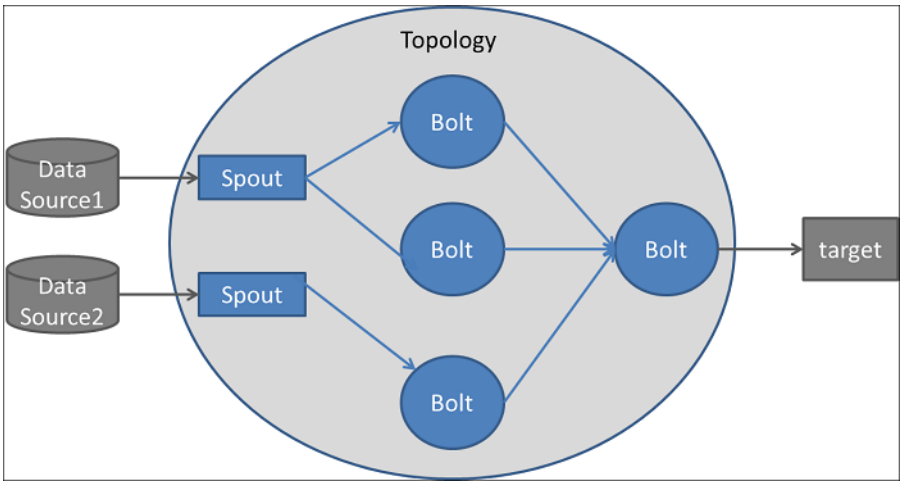

Fig. 1. Components of Strom Cluster

Storm creates what it calls as topologies to do the real time processing. First the topology will be created and then it will be feed into the storm processing engine. Topology is created with the help of a directed acyclic graph. The components of the topology are

(1) **Tuple**
—Tuple is the basic unit of the storm and it is the main data structure in storm.
—It can support all data types, usually it will be in the form of comma separated values

(2) **Stream**
—Stream represents unbounded sequences of the tuples.

(3) **Spouts**
—It is basically input source of the stream. By and large, spouts will read tuples from an outer source and discharge them into the topology.
—You can compose spouts to peruse data from data sources, for example, database, distributed frameworks, informing structures or message line as Kafka, from where it gets consistent data,

changes over the genuine data into a surge of tuples and produces them to bolts for real handling. Spouts keep running as errands in laborer forms by Executor threads.

(4) **Bolts**
—Bolts are the processing units in the topology they get the input data either from the other bolts or Spouts.
—This Bolts are distributed across the cluster by the storm. Bolts can do anything from simple transformations like filtering to complex operation like joins.
—To perform a simple join you may need just one or two Bolts. But to perform complex operations we may to pass information from multiple outputs to get the final desired result.

Storm distinguishes between the following three main entities that are used to actually run a topology in a Storm cluster:

(1) Worker Processes
(2) Executors
(3) Tasks

The following figure shows the relationship between this three entities.

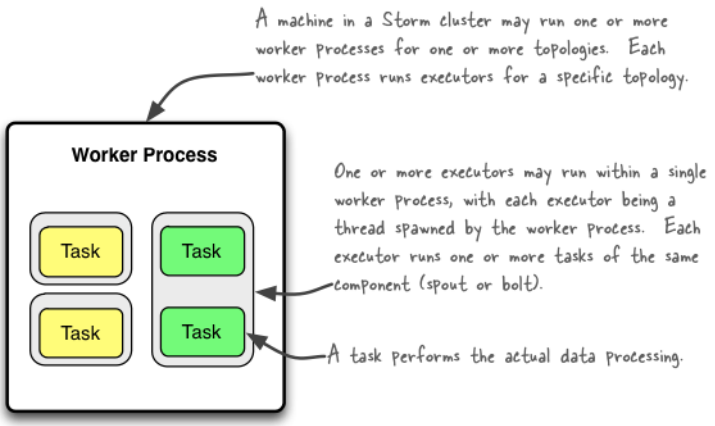

Fig. 2. Relationshp between workers processes, executors and tasks

(1) **Executors**
—Executors are the threads spawned by the worker process. Each executor can run one or more tasks.

(2) **Worker Process**
—A worker Process used to execute a part of the entire work. A worker process will choose execute one part of the topology with the help of the executors.

(3) **Tasks**
—This where the actual work is done. This is fundamental unit of the work that will be done in the full topology. So basically it can be a execution of a spout or bolt. At a given instance multiple threads can be executing a task.

In Storm the streams of data from one bolt to another. Streaming grouping protocols control how the tuples are being forwarded from one system to another. Following are four main grouping protocols used by the storm.

2.1.1 ***Shuffle Grouping***. In this grouping mechanism the tuples are transferred randomly to all the nodes across the cluster such that all of them will get equal quantity of workload. The following figure depicts this mechanism.

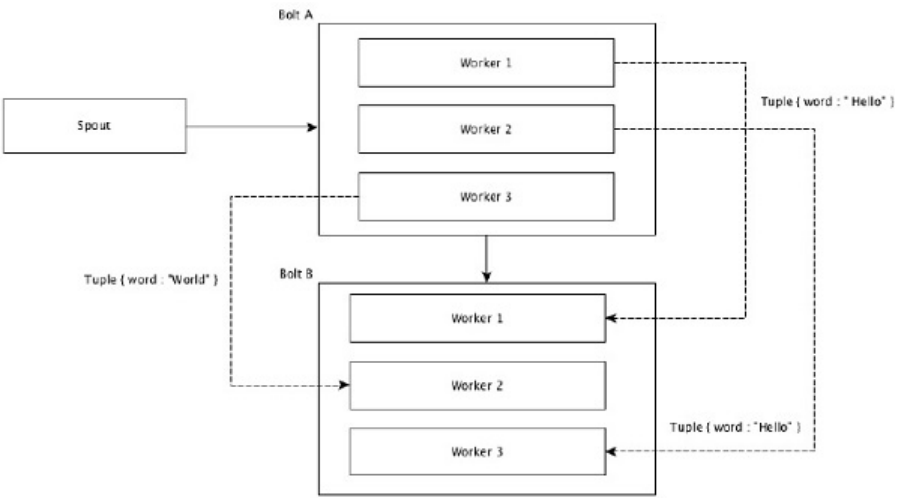

Fig. 3. Shuffle Grouping

2.1.2 ***Field Grouping***. In this grouping mechanism the tuples are grouped based on certain fields and others fields are ignored. All the tuples with same fields under consideration will go to same node in the system. For example if we are grouping based on the words, then all tuples which have common word will go to same node. They use consistent hashing to implement this. The following figure depicts this mechanism.

2.1.3 ***Global Grouping***. Here all the streams are grouped and sent to one bolt. Therefore this grouping sents all the generated tuples to single instance of a machine. The following figure depicts this mechanism.

2.1.4 ***All Grouping***. All Grouping sends a solitary duplicate of each tuple to all occasions of the getting jolt. This sort of grouping is utilized to send signs to bolts. All grouping is helpful for join tasks. The following figure depicts this mechanism.

2.2 Internal Design of Storm Cluster

There are two types of nodes in any Storm cluster. They are :

—Nimbus

—Supervisor

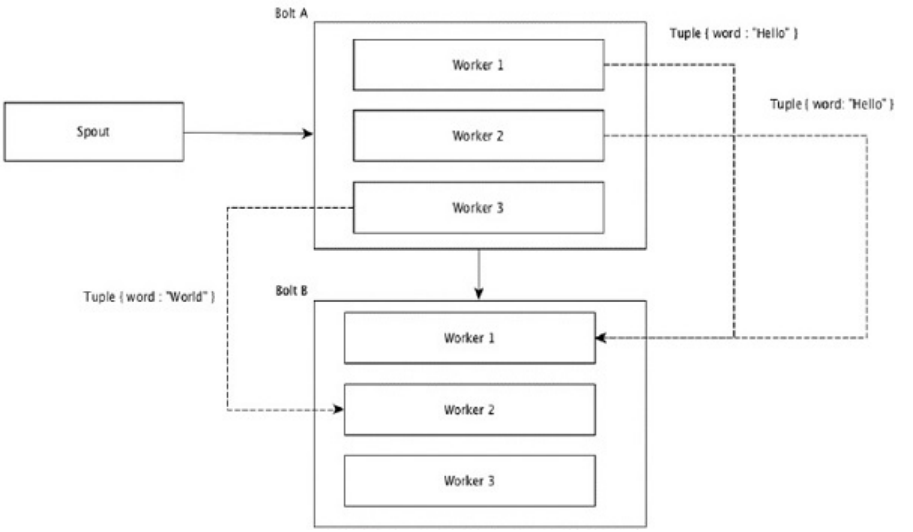

Fig. 4. Field Grouping

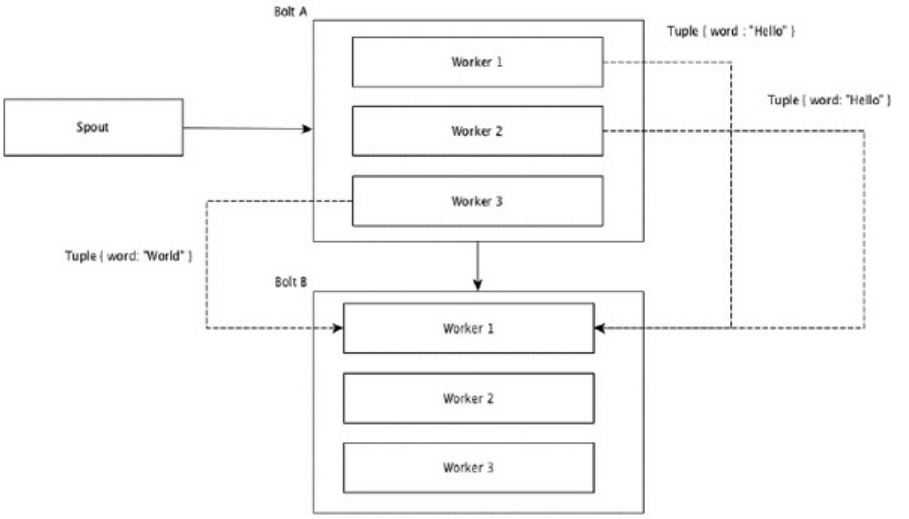

Fig. 5. Global Grouping

2.2.1 ***Nimbus***. Nimbus is also called as Master Node in storm Cluster. As a master node it is responsible for distributing the code across the cluster and assigning the jobs to all the other nodes by passing the required data and monitoring their failures and state. It is an Apache thrift service which enables the users to code in any programming language they want which makes it easier for the application programmer to develop applications. Nimbus relies on Zookeeper for monitoring the tasks of all the other worker nodes and returning their state make to the Nimbus Node, so that it can take appropriate decisions.

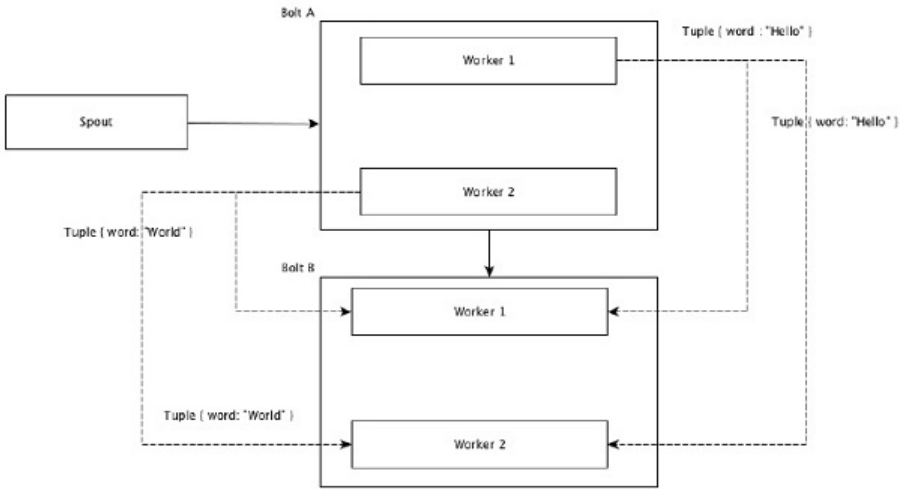

Fig. 6. All Grouping

2.2.2 ***Supervisor***. Supervisor has a set of Worker Nodes. It listens to the instructions given by the Nimbus, master node and makes the worker nodes do that work. Each of the worker process executes a part of the topology. We can think of the Supervisor as an intermediary between the worker node and Nimbus.

The Relationship between the Nimbus and the supervisor is depicted in the figure below

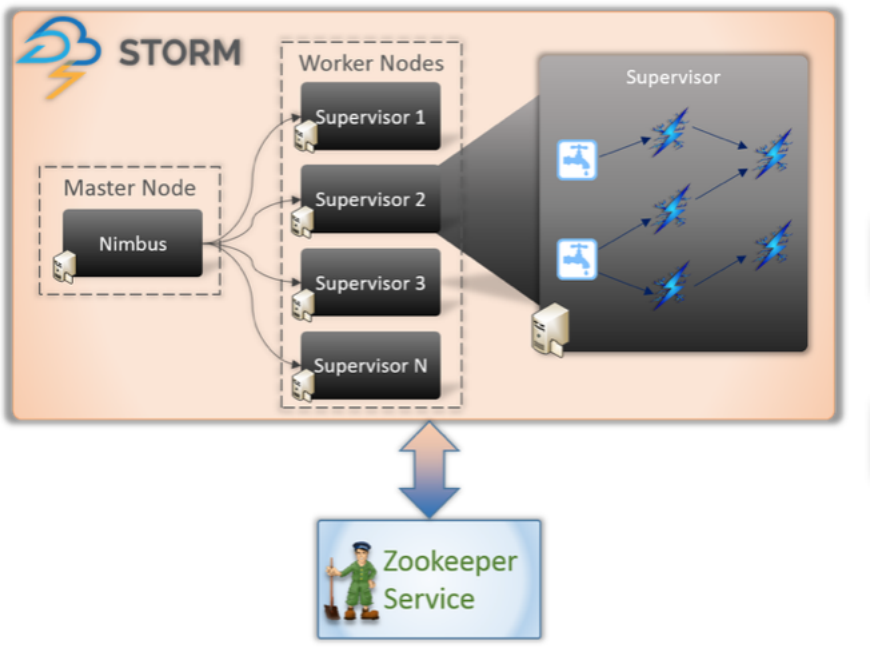

Fig. 7. Storm Cluster

2.3 Message Processing Guarantees

Every streaming system offers some sort of scheme by which it guarantees how the messages will be processed. Apache Storm also offers three such types.

(1) No Guarantee

(2) At least once Semantics

(3) Exactly once Semantics

(1) **No Guarantee** : It is the easiest semantics to satisfy. Here the system doesn't offer any guarantee and it ignores the tuples which may not get processed properly.

(2) **At least once Semantics**

—This means that when Tuple comes into the system, goes out at the spout, the framework guarantees you that this Tuple will not be missed. It will be processed. At least once.
—Storm handles this style of message processing guarantees by using Tuple trees, anchoring, and spout Replay.
—A tuple tree talks about a data structure handled by the framework. So the user doesnt necessarily see the Tuple Tree or do any interaction with it.
—Every Tuple that comes out of a spout becomes the main node or the root of a new tree.
—The first Bolt creates a set of Tuples based on its input.Each of these Tuples becomes nodes in this tree. And as each of these Tuples are again processed by other Bolts and create Tuples again, these Tuples are added to the tree. So the Tuple coming out of the spout becomes the root of the tree and it kind of keeps track of this Tuple by relying on this tree that the framework is taken care of.
—We can then give it a specified timeout. And if within that timeout period, this whole tree is not completely processed, the system considers some of these tuples lost.
—The way this whole mechanism works is relying on certain special tasks in the system called Acker tasks, that keep track of these tuple progresses. And this is called as Anchoring Mechanism.
—Not all of the spouts coming with Storm have that and not all of the infrastructure supports replay. So if you want to have your spout provide this reliability measure, you must make sure that the spout itself can do replay.

It is explained with help of following figure :

(3) **Exactly Once Semantics**

—Apache storm achieves the exactly one semantics with the help of Trident mechanism.,
—If you want to guarantee exactly once, you want to provide for cases when one machine processes the data and then after it processes a certain tuple it dies. If it has processed it once so you don't want to process it again.
—But the system has died, so when it boots back up and restarts, its memory's refreshed. And it doesn't have any notion of what it has done before. So here is a case where you have to store the state of your topology somewhere in a pre-existent storage, like a hard drive.
—Trident is used to store this state, but the way the state is stored is upto the user. Trident provides serveral High Level APIS and also some prebuilt connectors to various NO SQL stores.
—You can think this in terms of normal tranasaction in traditional database management systems. Due to the high overhead this mechanism is the slowest among all the three .

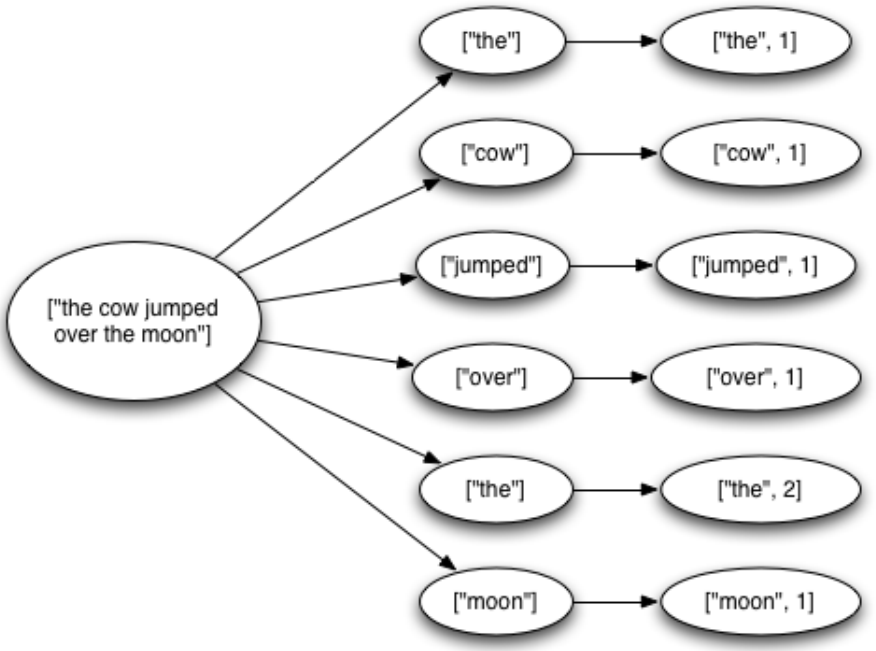

Fig. 8. Tuple Tree

2.4 Problems with Apache Storm

To provide fault-tolerant messaging, Storm has to keep track of each and every record. By default, this is done with at least once delivery semantics. Storm can be configured to provide at most once and exactly once. The delivery semantics offered by Storm can incur latency costs; if data loss in the stream is acceptable, at most once delivery will improve performance.

3. SPARK STREAMING

Spark streaming is gaining lot of attention and popularity in the open source community, and also in the big data enterprise computation industry. Spark Streaming provides stateful stream processing. So uses this we can maintain state without much overhead as we had Apache storm. Spark streaming can be considered as an extension to the Core Spark API. So, it provides all the functionality the core spark API provides, that includes high throughput, scalability and reliability.

3.1 Spark Streaming Concepts

3.1.1 ***Micro Batch Processing***. The main advantage of using Spark as a streaming system is it's ability to maintain state. As mentioned earlier even the Apache storm can do this but it has to store the state information again in some separate storage which incurs additional overhead. Spark streaming has a new way of doing this. It does this with help of something called as Microbatch processing.

And the whole idea is that you have RDDs, Resilient Distributed Datasets, that keep a lineage of how each partition of the data is created. And it will try to recreate data, run the computations again if there is a failure. So the idea behind Spark's framing is that we want to window a little bit of data. So

we take a sliding window, grab a little bit of data in a window, run that little window of data as a batch, as if it was a batch. And then repeat the process over and over and over again. The main idea here is called discretized stream processing. Spark streaming system calls these D-Stream, as in Discretized streams.

So instead of having the live data stream coming in, we chop them up into smaller batches of X seconds. And then, Spark treats each batch of data as, each little batch of data, as an RDD. And processes them using the RDD operations that we've talked about in previous videos. Then finally, the processed results of RDD operations are then returned in batches after batches after batches. So you can feed them into another streaming system or you can send them to HTFS, or whatever you want to use them. Again, the results are one batch of results, then another batch of results, then another batch of results. Batch sizes are very important in Spark streaming. The choice of batch size has a huge impact on what sort of performance you can get out of the system, what sort of latency and what not. Practical latencies for Spark streaming are in the range of one second and up. The best it can really do is about 300, 400 milliseconds. Any lower than that, and the overhead of each batch just completely overwhelms the system.

So basically if we are designing a system that requires very fast responses, low latency to your events, Spark streaming is not the solution. If we want an event to come in and be processed within 10 milliseconds, one millisecond, maybe sometimes you need even faster, maybe sometimes if we are fine with a little bit larger, but not as much as a second, Spark streaming is not the system. However, if the system that we want to design if the application that we are planning to build, can handle one second of latency, even better, if it can handle 10 seconds of latency, so there's a new batch of results every 10 seconds, then Spark streaming could be a very effective tool.

SPARK streaming comes with a couple different already built-in sources of data that can connect to different sources and extract data from them. Kakfka is one the most popular tool among them.

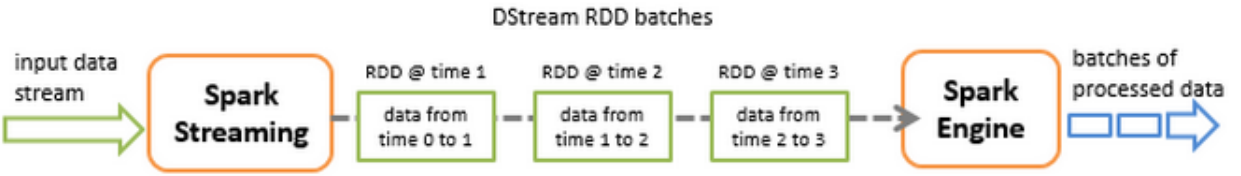

Fig. 9. Spark Streaming Pipeline

3.1.2 ***State Information with RDDs***. As we have seen with Apache Storm we need to store state information in some external database to maintain the state. But for this we have to keep the connection throughout and it will cause additional overhead and induces additional latency. Spark streaming provides a better way to handle this.

As mentioned before it's hard to manage state in a streaming system, because if there's a failure, if we manage the state inside the processing nodes in Storm or something like that, then the whole state can go down. If you want to connect to an external system like Redis, Cassandra, database, like for example, Trident does,you have the overhead that you have to pay for.

What Spark streaming provides you, and this is a side effect of its microbatch nature, is that it allows you to keep states inside the processing nodes. And if there is a problem with the processing node, if the processing node goes down, it can automatically bring up the states again. So the way that it does

this is by giving you a function updateStateByKey. This way, you can maintain arbitrary state while continuously updating it. And if there's a failure, the system will automatically get it back for you. Now, how to use this is that you define the state. So the state can be any arbitrary data type. It can be a map of maps of queues or whatever. So it's basically any data type that you want can be your state. And then you also need to define a state update function which the state update function specifies the function, how to update the state using the previous state and the new values for an input stream. Now what happens is that Spark streaming, every time it is processing one little batch, one little batch of the D-stream, one second worth of data, it calls the state update function and applies it to the data type that you define as the state in every batch for all existing keys. Now, if there is an issue with the system, so you can imagine, okay, each of my processing nodes in the cluster are keeping the state by having this. Now if there's an issue and something goes down, the state is now part of an RDD, so the system automatically keeps the lineage of it, and frequently makes a checkpoint of it. So if there's a failure, it can load the checkpoint and then rerun everything that was run after that checkpoint up to the last point that it broke down. So this way, you can keep arbitrary state and try to get closer to that holy grail of kappa architecture, knowing that this only works when you are not that much in need of fast processing, in need of low latency. More about the Kappa architecutre will be explained in later sections.

Spark Streaming gives us a huge bandwidth. It can actually process huge amount of data in a given time, more than Storm. But if we are limited by the latency, then of course we are limited to not using Spark Streaming.

3.1.3 ***Ease of Use***. As mentioned earlier Spark uses mechanism of Micro Batch Processing to provide streaming. This is done with the help of RDDs. What this means it also allows use to all batch processing features of the Spark. It can, each of the discretized streams, is an RDD by itself. So you can do whatever you could do in regular Spark with them.

Aside from the niceties of updates state where keeping this state and everything, is that it allows us to get in touch with a rich ecosystem of other big data tools that are now part of the Spark System. We have access to Spark SQL where you can just use SQL statements to query data. You can access Spark ML, Machine Learning Library, and Machine Learning to do ML, machine learning, on your little bit of streaming data that are batched in, micro-batched in, let's say, one second worth of data. You can access to graphics, SparkR, the R language for statistical computation. Again, the disadvantage is that it's not really streaming in the strict sense. So it's really batches data, and then runs that batch of data very quickly.

3.2 Problems With Spark Streaming

So Spark has lot of good features which makes it very popular and makes it first choice for many business streaming needs. But there are some downsides of using Spark streaming too. As mentioned earlier spark uses micro batch processing to enable stream processing and it maintains state information and all the other features that spark provides with the help of RDDs. But the downside of it is we are actually not doing streaming because we are not processing the events as soon as they arrive instead we are waiting for a fixed window and loading the tuples in that window and processing them as a batch. If we keep this window time very low it severely affects the performance of the entire system. But if we increase it then we are inducing latency. So there is a trade off between them.

So basically if we are designing a system that requires very fast responses, low latency to your events, Spark streaming is not the solution. If we want an event to come in and be processed within 10 milliseconds, one millisecond, maybe sometimes you need even faster, maybe sometimes if we are fine with a little bit larger, but not as much as a second, Spark streaming is not the system. However, if the

system that we want to design if the application that we are planning to build, can handle one second of latency, even better, if it can handle 10 seconds of latency, so there's a new batch of results every 10 seconds, then Spark streaming could be a very effective tool.

4. APACHE STORM OR APACHE SPARK ?

In the above sections both the working of Apache storm and Apache Spark is explained in detail. As we have seen both the frameworks have their own advantages and disadvantages. So choosing which bettiondepee ernds u pon thi and the type of the business application.

4.1 When Apache Storm is Preferred ?

Apache storm can be preferred when the latency is very important factor rather than maintaining the state. Even though Apache storm can maintain it is induces additional overhead because we have to store that state in some external database and the connection should be maintained throughout and it induces lot of latency. So it can be used where there may be some scope of losing or not processing few tuples. Instead of forcing storm to maintaining this exact state semantics, we can use additional batch processing along with the storm which can be used to update the information later. This makes storm an important component of Lamda architecture which we explained in detail in later section.

4.2 When Apache Spark is Preferred ?

Spark streaming can be preferred if the system can tolerate small latency. If it can tolerate that small latency then it can use all the benefits provided by spark like maintaining the state information and all the features of the Spark API.

If we want an event to come in and be processed within 10 milliseconds, one millisecond, maybe sometimes you need even faster, maybe sometimes if we are fine with a little bit larger, but not as much as a second, Spark streaming is not the system. However, if the system that we want to design if the application that we are planning to build, can handle one second of latency, even better, if it can handle 10 seconds of latency, so there's a new batch of results every 10 seconds, then Spark streaming could be a very effective tool.

5. STREAMING ARCHITECTURES

There are various different Architectures which can be used to satisfy the needs of modern day real time processing requirements. Each has it's own advantages and disadvantages. But all of them try to satisfy the same requirements.

The most clear of these prerequisites is that data is in movement. At the end of the day, the data is consistent and unbounded. It's really about when you are breaking down this data that issues. In the event that you are searching for answers against the ebb and flow depiction of data or have particular low-inertness necessities, at that point you're likely taking a gander at a real-time situation. Moreover, there are all the time business due dates to be met. All things considered, if there were no results to missing due dates for real-time investigation, at that point the procedure could be batched. These results can extend from finish inability to just debasement of administration.

Since we are discussing huge data, we likewise hope to push the cutoff points on volume, speed and potentially even assortment of data. Real-time data handling regularly requires characteristics, for example, versatility, blame tolerant, consistency, strength against stream imperfections, and should be extensible.

Out of may such architectures two of them are explained in below sections. They are two of the most popular streaming architectures used by people and each of them can be implemented using various tools and frameworks.They are :

(1) **Lambda Architecture**

(2) **Kappa Architecture**

5.1 Lambda Architecture

Lambda Architecture is a Streaming architecture that is designed to handle massive amounts of data by taking advantage of both the batch and the stream processing methods.

Lambda Architecture is when you want to do of course real time processing of events that come into your data pipeline. The way they architect this system is by having two parallel data processing paths. The first processing path would use a stream event processing system like storm that was explained earlier. Instead of storm we can also use other streaming systems like Spark streaming. Then on the parallel path you have the batch processing system, like Hadoop.

5.1.1 ***Need of Lambda Architecture***. The reason that people do lambda architecture is because things In a computer system fail all the time.Batch processing can handle failures just fine. Now of course, this is an oversimplification. There are many different failure models, but Batch processing is designed around this whole idea. I have a batch of data to process, if my machines fail, something in the distributive system goes wrong, there is a missing heartbeat at some point, so if i restart the computation everything just works just fine.

But in streaming system it's not the case. Here we want the events to be processed as soon as they arrive. But the problem comes when some of these events fail. Then in this system the event is gone. Lambda architecture tries to fix this issue. They agree that maintaining state and making the streaming system fault tolerant is difficult. So they have back processing pipeline that will be running in the background for every 24 hours or so. Then the data from both will be synchronized as well as the state of both the real time streaming system as well as the batch processing system.

So in this way we can use both the advantages of batch and streaming systems to get good results. The details of this Architecture along with it's different layers and components is given below.

5.1.2 ***Components of Lambda Architecture***. There are three main components in the Lambda architecture. They are:

—Batch Processing Layer

—Real time or Speed Layer

—Serving Layer

The input to the both the processing layers is same. The functionality of each of this three layers is described below.

(1) **Batch Processing Layer** Batch processing aims to compute the pre computed views that have to be served. It tries to make sure that gives accurate results despite of high latency. It can handle huge quantities of data and at a reasonably good speed it gets accurate results from them. It is fault tolerant and highly scalable.
It takes input from the normal source and then uses to update or sometimes correct the existing views thereby makes sure that we eventually see the correct results. The output of this layer will be a read only database.
Normally Apache Hadoop is used for this purpose.

(2) **Speed Layer** This layer processing the input streaming data in real time and doesn't care much about the fixups. It aims to reduce to latency so it doesn't care much about the throughput and the fault tolerance as this will be taken care by Batch processing layer eventually.

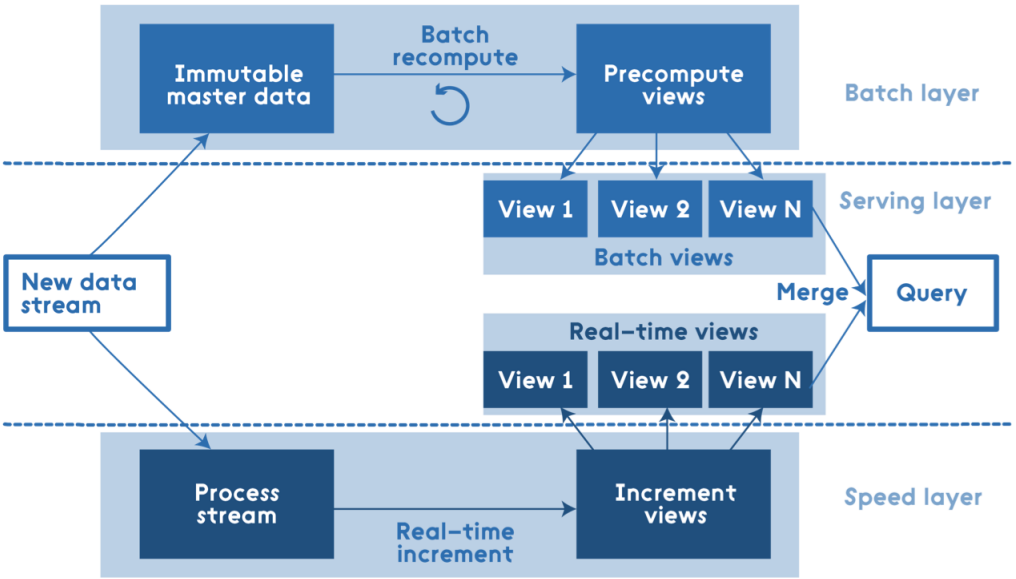

Fig. 10. Lambda Architecture

So basically we can think of speed layer as the one which provides temporary intermediate views to the user. The information provided by the Speed Layer may not be accurate but it provides the information fast so that user will see some meaningful information in real time.
Some commonly used mechanisms for this purpose are Apache storm and Spark Streaming.

(3) **Serving Layer** The output of the speed and the batch layer is handled by the serving layers. It has views and it responds to the user queries using this views.
Usually people use Druid which can provide a single unique cluster to handle both the layers. Dedicated stores such HBase, Cassandra are used to get the output from the speed layer, whereas Apache Hive, Apache Pig, Impala are used to get the output from that Batch Processing systems.

5.2 Kappa Architecture

Kappa architecture is a simplification of Lambda Layer. It removes the batch processing layer in the Lambda architecture and just relies on real time streaming layer to serve the views.
The Architecture of the model is depicted in the figure below.

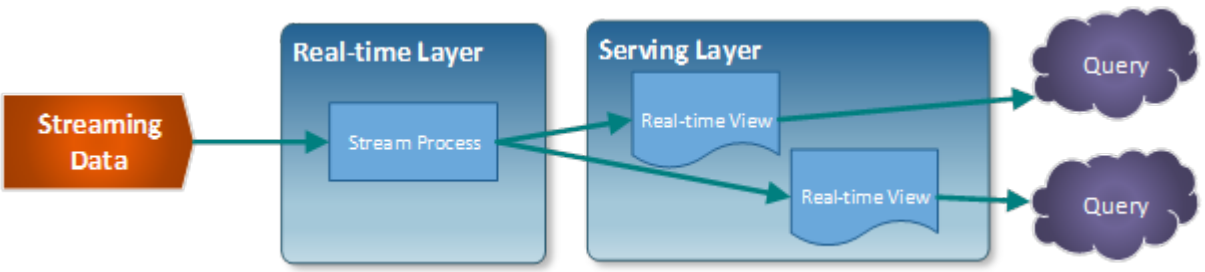

Fig. 11. Kappa Architecture

5.2.1 ***Methodology***. Kappa Architecture was first introduced by Jay Kreps. Here we just try to do the entire work using the streaming layer without the help of batch processing layer. So it is by no means a replacement of Lambda architecture except for some particular use cases. In this architecture we just use the results from speed layer to serve the user queries.

The main idea is to do the work of both the real time processing layer and the batch processing layer in single processing engine. So here even the reprocessing occurs in the same engine. To make this happen we need the incoming data to be replayed quickly. This may be entire data or data from particular point. So if there is any change in the system then the second stream can replay the data again and complete the processing and update the results.

So it attempts to simplify by using just one code base rather than maintaining a separate code base for batch processing and real time processing results. Also for serving the final results the serving layer doesn't have to check in two results and merge them as there is only one layer it can directly display the results so it will have less overhead and it will be faster compared to lambda architecture.

But the major complication here lies in the stream processing engine. As it holds the responsibility for maintaining the state information replaying the input stream if necessary and maintaining the order information which can be handled easily by batch processing systems.

Typically Spark Streaming processing engine would be a good choice here because of mechanism of processing the data in the form of micro batch mechanism.

6. LAMBDA OR KAPPA ARCHITECTURE ?

Some real-time utilize cases will fit a Lambda architecture well. The same can't be said of the Kappa Architecture. On the off chance that the batch and streaming processing results are indistinguishable, at that point utilizing Kappa is likely the best arrangement. Now and again, in any case, approaching an entire set of information in a batch window may yield certain improvements that would improve Lambda performing and maybe significantly easier to actualize.

There are additionally some exceptionally complex circumstances where the batch and streaming calculations deliver altogether different outcomes (utilizing machine learning models, master frameworks, or naturally extremely costly activities that must be performed contrastingly in real-time) which would require utilizing Lambda.

In this way, that covers the two most mainstream real-time information handling architectures. The following articles in this arrangement will plunge further into each of these and we'll talk about solid utilize cases and the advancements that would frequently be found in these architectures.

7. STREAMING ECOSYSTEM

In the final section the entire Streaming Ecosystem will be explained in brief. All the pieces of the streaming ecosystem will be put together to get the full picture. The end to end description of things will make it easier to understand the full work flow.

There are so many tools and frameworks to pick from while implementing an architecture. Each of these tools have their strengths and weaknesses, and it's best to pick and choose and use the right tool at the right time in the data processing pipeline. And not try to beat a framework to do something that it's not quite optimized to do for it. But basically the entire streaming ecosystem can be divide into three parts. They are

—Stream Sources

—Processing Systems

—Output Sinks

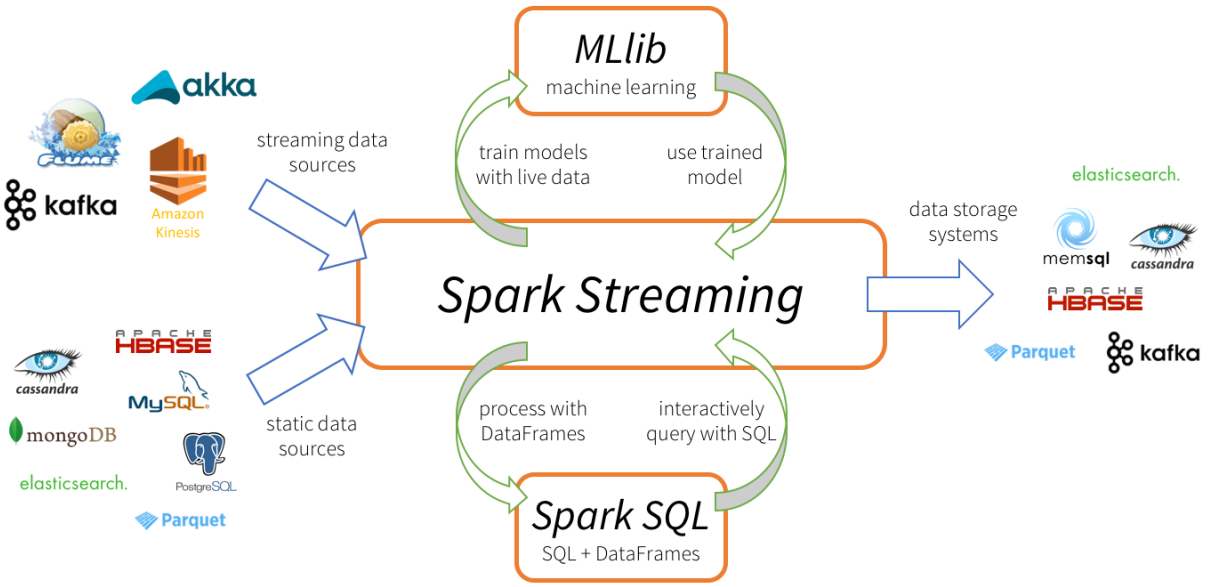

Fig. 12. Spark Streaming Ecosystem

To illustrate this ideas Spark streaming ecosytem is taken as example here. Even though it is not completely relevant to other Streaming ecosystems. The ideas that will be explained here can be easily extended to any other ecosystem.

With respect to Spark Streaming ecosystem the components of it are explained below.

(1) **Input Stream Sources**

—Here we want the system to gather the data from a large set of users.They could be all over the planet.

—So you want to be able to gather the data. So you really want a good distributed funnel sort of mechanism.

—The system should be fault tolerant, highly scalable and should give very high throughput.

—So Kafka is a very popular example, and it would be perfect for this sort of use case. Kafka has producers, consumers, and then a whole bunch of brokers which synchronize with each other using ZooKeeper. Kafka can provide high throughput cluster of couple Kafka nodes, three, four, five Kafka nodes can handle huge amounts of incoming traffic and putting them, storing them in discs and then So to make sure that it kind of buffers the ups and downs, and the networks, and noise. It was originally developed by LinkedIn, but now it's an Apache open-source project. There are a whole bunch of tools that you can interact with Kafka. There are a whole bunch of command-line tools. kafka-console-producer, consumer, topics. And then there's some recent efforts to provide graphical interfaces for Kafka. There's a Kafka-manager now.

—Data sources can be of two types. They are :

1. Static Data Sources

2. Streaming Data Sources

—Some of the examples of static data sources are Cassandra, Apache HBase, MySQL, Postgre SQL, MongoDB

—Some of the examples of the streaming data sources are Apache Kafka and Apache Flume.

—The best input source varies from application to application and has to be choosen according to the need.

(2) **Processing system**

—After gathering the entire data and after setting up the input data source we need some processing system to process the data and give some useful information from that.

—The reuirement of this is it should be distributed, high speed, fault tolerant, scalable and sometimes we need to preserve the state of the system also

—Here we can explained two of the most popular such systems they are Apache storm and Spark streaming.

—In context to Spark streaming it also enables to query external databases with the help SPARK SQL and also enables us to machine learning with the help of MLlib library.

—This is responsible for processing data and maintaining it's state and giving the final desired result to the application

(3) **Output Sinks**

—After getting the required results from the streaming data the final output will be consumed by this sinks.

—They can be used in various ways. In some applications we directly display the results whereas in some applications we pass this output as input to one more streaming system

—Some of the Common data storage systems are, Apache HBase, Apache Kafka, Cassandra, Redis.

8. CONCLUSION

In this Paper, various technologies and architectures related to real time processing have been explained. Along with traditional batch processing real time processing has become a must for many business organization today. Right from big social media websites to e commerce websites where every action of the user will be tracked and processed in real time to provide the user with better experience and in many to track fraudlent and suspicious activity we need to process the actions of the user in real time. We also saw that we need new processing systems and architectures for this purpose as tradition methods such as batch processing won't be able to give us the response in real time. Few such technologies such as Apache Storm and Spark streaming have been explained. And we also saw that both this systems have their own advantages and disadvantages and have to be used based on the need. Few architectures like Lambda and Kappa are also discussed. Finally a general components of the entire Streaming Ecosystem have been explained to get the full picture of the entire Work flow.