HOW DOES SPARK STREAMING FIT INTO THE EIGHT REQUIREMENTS OF REAL TIME STREAM PROCESSING

Merlyn Mary Michael[1], Jayakrishna V[2]
Amal Jyothi College of Engineering, Kerala, India
Email: merlynmarymichael@cs.ajce.in[1], vjayakrishna@cs.ajce.in[2]

Abstract

The demand for stream processing is increasing these days. The reason is that, organizations need to process data fast to react to changing business needs and conditions on real time. The value of stream processing systems mainly come from the timeliness of the results that they can provide. Today, stream processing finds its application in almost every industry - wherever stream data is generated through human activities, machine data or sensors data. Many distributed stream processing frameworks are currently available. This paper evaluates the stream processing framework Spark Streaming to identify how well it fits into the eight requirements of real time stream processing proposed by [3].

Index Terms: micro batches, real time, stream processing, Spark Streaming.

I. INTRODUCTION

Big data is a very popular term these days. It is mainly characterized by the three V's - Volume, Variety and Velocity. Volume refers to the quantity of data. Variety refers to the range of data types and sources. While velocity determines how fast the data is generated and processed to meet the demands. Stream processing is about the velocity aspect of big data. Some business problems that can be solved using Stream processing are fraud detection, pricing and analytics, intelligence and surveillance. The streams of data are accumulated from different sources and are considered most valuable when they arrive. So it makes sense to process these data as soon as they arrive using real time streaming analytics.

II. BACKGROUND

Many stream processing frameworks are currently available. From the Apache landscape three major frameworks that are available for stream processing are Spark Streaming, Storm and Samza. Their major traits are summarized in the Table I.

A. Spark Streaming Model

In general Spark Streaming model works as follows:

Spark Streaming can ingest data from multiple sources.

Features	**Real Time Stream Processing Systems**		
	Spark Streaming	**Storm**	**Samza**
Stream Processing Model	Microbatching	Native	Native
Programming Model	Declarative	Compositional	Compositional
Data Guarantees	Exactly once	Atleast once	Atleast once
Maturity	High	High	Medium
Latency	Medium	Very low	Low
Throughput	High	Low	High

Table. I

like Kafka, Flume, Kinesis, Twitter or even tcp sockets. The data is then processed using the high level algorithms. And the processed data is finally pushed to dashboards, filesystems or HDFS. Spark Streaming model is illustrated in Fig.1.

Spark Streaming uses a microbatching method for processing data streams. As data arrives, it forms microbatches over intervals of time. Each microbatch forms an RDD, the major abstraction in Spark. RDD is an immutable distributed collection of objects. RDDs are partitioned and computed across multiple nodes in a cluster in parallel. RDD is the concept of Spark that supports the main features of failure recovery, scalability load balancing.

B. Spark Streaming Components

The architecture of Spark Streaming is shown in Fig.2.

The components are briefly described below: A Spark Streaming application will have a *driver* program that has the main method. A *Streaming Context* is an object defined to specify the processing to be applied to the streams of data that continuously arrive.

Driver: Connects to the cluster manager to request for resources. It is also responsible for creating DAG of tasks corresponding to streaming job and submitting it to the cluster for execution.

Cluster Managers: Spark can work with a Standalone cluster manager, Mesos or YARN. They are responsible for allocating resources for execution.

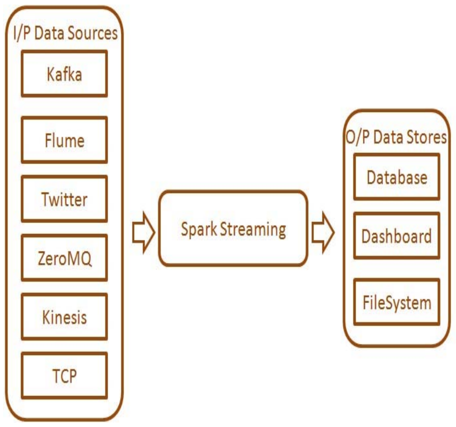

Fig. 1 Spark Streaming Model

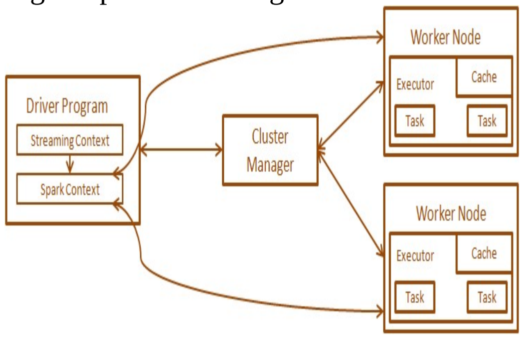

Fig. 2 Spark Streaming Architecture

Workers: These are the nodes where the execution happens. They have executors that execute tasks and have cache to support the task execution.

Receiver: In a Spark Streaming application the major component is the receiver, that is a task that runs throughout the application life time, to receive streams of data. They run within an executor in a worker node.

The following sections describe each of the eight requirements of a Real Time Stream processing System [ref] and illustrates the features of Spark Streaming that best fits into those requirements. The features of Spark Streaming are assessed in terms of the version 1.6.

RULE I

The first requirement for a real-time stream processing system is to process messages "in-stream", without any requirement to store them to perform any operation or sequence of operations. Ideally the system should also use an active (i.e., non-polling) processing model [3].

This requirement suggests that a real time stream processing system should be an active system that is data/event driven and should avoid storing data to disk before the data is processed. This is because accesses to disk can introduce unwanted latencies in the processing, which would badly impact the timeliness of the results. Spark streaming achieves this goal by making use of in-memory computations in a large cluster in a fault tolerant manner. It uses an abstraction called RDD for this purpose. an RDD is a read-only, partitioned collection of records. RDDs are fault-tolerant, parallel data structures that allows users to explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators. This in memory computations offer Spark Streaming a great speed up and allows it to deliver results with minimum latency.

Streams of data can be processed in two ways. One is the native method while the other is the micro batching method. In native method each record or event is processed one by one as it arrives in the stream.

While in microbatching method, streams of data that arrive are grouped to small batches over predetermined batch intervals. All the data that form a microbatch are processed together. Spark Streaming uses the microbatching method, where an RDD is formed out of a microbatch of data and is partitioned across the cluster of machines to be processed in parallel (in-memory). Thus in Spark Streaming the real time data streams are converted to DStreams (discretized streams) and processed. These DStreams are in turn a collection of RDDs.

RULE II

The second requirement is to support a high-level “StreamSQL” language with built-in extensible stream oriented primitives and operators [3].

This requirement suggests the need to have a high level language (with a rich set of stream specific operators) for processing continuous streams of data. This would greatly reduce development cycles and maintenance costs for streaming applications.

Spark streaming which is an extension to the Spark Core provides high level APIs in Java, Scala, Python and R. In general it provides a Declarative API with high order functions or operators. This makes programming easier, as the programmer does not have to deal with the low level details as the system would internally create and optimize the topology.

Spark Streaming handles data streams as DStreams (discretized streams). And it supports a rich set of transformation and output operations that can process or modify the DStreams. Some common transformation functions available on DStreams are map, flatMap, reduceByKey, cogroup etc. A number of output operations are also supported on DStreams. Output operations trigger the actual execution of all the DStream transformations and allows DStream’s data to be pushed out to external systems. Some examples of the output operations are print, saveAsTextFiles, saveAsHadoopFiles etc. Spark Streaming also provides windowed computations, which allows to apply transformations over a sliding window of data. the concept of windowed operations can be visualized as in

Fig.3.

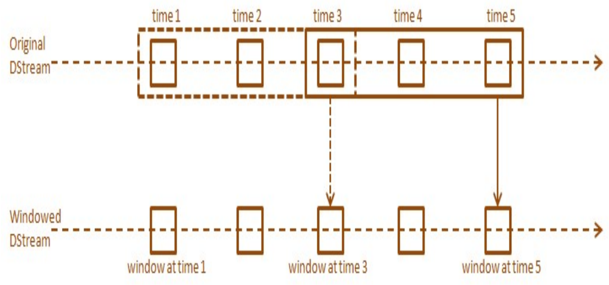

Fig. 3 Windowed Operations

Some commonly used window operations are reduce By Window, reduce By Key And Window.

In addition to the transformations and output operations listed above, Spark provides an additional module Spark SQL for structured data processing. There are a number of ways to interact with Spark SQL, which include SQL, Data Frames API and Data Sets API. All these interactions work on the same optimized Spark SQL engine. SQL interface allows to interact with datasets using basic sql syntax or hiveql queries. Data Frames API allows access to data in form of data frames, that organize distributed collections of data as named columns. Data Sets API is a new experimental addition in Spark 1.6.

Thus Spark streaming with its API and Spark SQL provides an extensible set of operators that can process stream data to retrieve intelligence from it.

RULE III

The third requirement is to have built-in mechanisms to provide resiliency against stream “imperfections”, including missing and out-of-order data, which are commonly present in real-world data streams [3].

The two main notions in time in streaming are: event time and processing time.

Event time: is the time at which an event happened or the event was created in the real world. Usually a time stamp is encoded with the data when it is created.

Processing time: is the time measured by a machine that runs the stream processing application to process the event

Mostly in networks there will be a difference between event and processing times, introduced due to many reasons, like network delays, data rate spikes etc. This causes out of order data. As in real-time systems, data is never stored, the infrastructure must make provisions for handling data that is delayed, missing, or out-of-sequence. Spark currently does not support this feature but its future version will come up with the capabilities to handle event time and out of order data. It will give support for the concepts of 'event time' and 'slack' durations to help manage stream imperfections.

RULE IV

The fourth requirement is that a stream processing engine must guarantee predictable and repeatable outcomes [3].

Traditional stream processing systems use a continuous operator model. But in that model its difficult to handle features like quick fault recovery, load balancing etc. Spark Streaming uses a different model wherein it structures computations as a set of short, stateless deterministic tasks on RDD partitions that are distributed across the cluster. DStreams and RDDs track a lineage for fault recovery and for handling stragglers. A lineage is basically a graph of deterministic operations used to build a DStream/RDD. Lineages are tracked at the level of partitions. Thus if a node fails, the RDD partitions that were running on it can be rebuilt by using the information in the lineage. That is in case of failures the partitions on the node will be recomputed by re running the tasks in the lineage, on the original data (of the partition).

For stateful operations checkpointing is enabled by default. When checkpointing is enabled, lineages are not allowed to grow indefinitely. This is done by removing the parts of the lineage once checkpointing is done.

Thus with the features of lineages and checkpointing Spark Streaming ensures predictable and repeatable outcomes.

RULE V

The fifth requirement is to have the capability to efficiently store, access, and modify state information, and combine it with live streaming data. For seamless integration, the system should use a uniform language when dealing with either type of data [3].

A very common application for this would be in fraud detection, in transactions. For doing this effectively, the usual transaction pattern should be learned and stored as a signature and when a new transaction arrives on real time, it has to compared against the stored signature. If a difference is detected during the comparison, a fraud is detected. For doing this in real time, stream processing systems should be capable to well integrate with stored data.

Spark Streaming integrates well with Hadoop and most of the available NoSql stores. The main programming abstraction in Spark Streaming, the DStream or RDDs allow batch and streaming workloads to interoperate seamlessly. Users can apply arbitrary Spark functions on each batch of streaming data: for example, an RDD can be formed from a static data set (say a file on HDFS) and this can be easily joined with a DStream using the 'join' operator, as shown in the example:

```
val filerdd = sparkContext.hadoopFile("file to
be joined") kafkaDStream.transform
{batchRDD=> batchRDD.join(filerdd).filter(..)}
```

Another common example found in production is Spark Streaming's integration with the Cassandra store.

DataStax has provided a Spark-Cassandra connector that can enable Spark Streaming to effectively communicate with Cassandra. This connector handles type conversions between the two and also internally aligns Spark partitions with that of Cassandra partitions to get great performance for reads and writes. The configuration parameters of the connector, Cassandra and Spark Streaming should be intelligently tweaked to get the desired performance for the application.

Spark's concept of RDD and microbatching helps it very well integrate streaming and batch data for real time analytics.

RULE VI

The sixth requirement is to ensure that the applications are up and available, and the integrity of the data maintained at all times, despite failures [3].

One major advantage of Spark Streaming is that it provides strong fault tolerance. As long as the input data is stored reliably, Spark Streaming will always offer "exactly once" semantics. (i.e., as if all of the data was processed without any nodes failing), even if workers, driver or receiver fails in between.

Spark Streaming ensures its exactly once semantics, by supporting driver, worker and receiver fault-tolerance.

A. *Driver Fault tolerance*

In order to ensure that the driver program is up and running all the time, check pointing should be enabled. Check pointing allows to save data periodically to a reliable storage like HDFS.

Check pointing keeps track of the state of the program, so if it crashes in between, Spark Streaming can read how far the previous run of the program got in processing the data and take over from there.

When Spark Streaming works with the Standalone cluster manager, we can make use of the --supervise flag to ensure that the driver program restarts automatically if it crashes.

B. *Worker Fault Tolerance*

For worker fault tolerance Spark Streaming utilizes the RDD lineages. In between if a worker node fails, the RDD partitions on it can be recovered by using the information in the lineage. That is all the tasks in the linage will be re run on the original RDD partition data to get back to the RDD state at which the node had failed.

C. *Receiver Fault Tolerance*

In Spark Streaming a receiver is a long running task, that keeps receiving the data from the input source. So fault tolerance of the receiver is important for ensuring zero data loss. Receivers can be reliable or unreliable. reliable receivers are the ones that can acknowledge the data it has received. reliable receivers are used in combination with reliable senders. Reliable receivers ensure zero data loss and exactly once semantics by receiving the data, replicating it and then acknowledging it back. Once the data is acknowledged by the receiver, the sender can update its offsets, to point to the next data to be retrieved.

Thus with the driver, worker and receiver fault-tolerance capabilities, Spark Streaming framework is highly available and it ensures 24/7 operation for stream processing applications.

RULE VII

The seventh requirement is to have the capability to distribute processing across multiple processors and machines to achieve incremental scalability. Ideally, the distribution should be automatic and transparent [3].

This requirement states that real time stream processing systems should be capable to scale well to any number of machines and should have the ability to automatically load balance across the available machines.

In stream processing applications, controlling the ingestion rate would not be sufficient to handle variations in data rates over extended periods. A viable solution for it is to allow the cluster resources to dynamically scale as per the processing demands.

Spark Streaming currently does not support automatic dynamic scaling of cluster resources, where resources are dynamically acquired and used based on higher processing needs.

But as Spark inherently divides a large job into smaller tasks for execution, this feature can easily redistribute tasks to a larger cluster if more nodes are acquired from the cluster manager. That is, one benefit of writing applications on Spark is its ability to scale computation by adding (manually) more machines and running in cluster mode. And Spark Streaming provides high level APIs with which users can rapidly prototype applications on smaller datasets locally, and use the unmodified code on even very large clusters.

RULE VIII

The eighth requirement is that a stream processing system must have a highly-optimized, *minimal-overhead execution engine to deliver real-time response for high-volume applications [3].*

Apache Spark is a cluster computing platform designed to be *fast* and easy for use. In terms of speed it is a great improvement over Map Reduce. Spark has the speed advantage because of its in memory computations, that allow data to be persisted in memory in between computations. This avoids the latencies involved in accessing disks in between. Spark's in memory computations makes it well suited for interactive and iterative processing also where Map reduce shows a poor performance.

The Spark ecosystem contains multiple closely integrated components. The Fig.4. shows spark ecosystem.

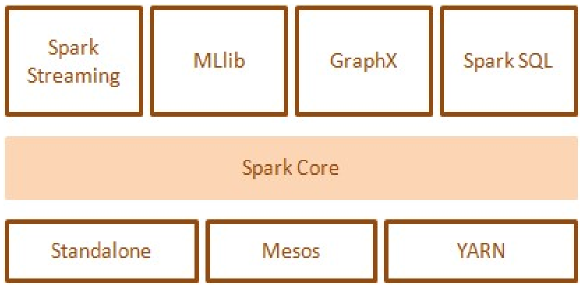

Fig. 4 Spark Ecosystem

At its core, Spark is a "computational engine". It is responsible for scheduling, distributing, and monitoring applications consisting of many computational tasks *over a cluster*. Spark provides a number of high level components involving different work loads. The different high level components are:

- *Spark Streaming:* for stream processing for real time analytics with minimum latency
- *MLLib:* for machine learning
- *GraphX:* for graph processing
- *Spark SQL:* used for structured data processing

Thus Spark provides a unified stack wherein all these components are designed to work together. The unified stack provides the following advantages:

1) all higher level libraries can take advantage of the features, improvements and optimizations of the lower layer (Spark Core) very easily.
2) the costs (for deployment, maintenance, testing, support, and others) are reduced, as a single system is used instead for separate software.
3) applications can be build that can seamlessly combine different processing models

In order to get the best performance from Spark, major factors to be considered are:

A. *Level of Parallelism*

This should be configured in a way that utilizes the available resources effectively. If there is too little parallelism, Spark may leave resources idle. If there is too much parallelism, the overheads associated with managing parallelism may build up and become significant.

B. *Serialization***:**

When Spark has to transfer data over the network or spill data to disk, it needs to serialize objects into a binary format. This comes into play during shuffle operations, where potentially large amounts of data are transferred. By default Spark uses Java's built-in serializer. Spark also supports the use of Kryo, a third-party serialization library that improves on Java's serialization by offering both faster serialization times and a more compact binary representation. If huge data sizes are to be handled, Kryo serialization may be better as it can reduce the amount of data transferred over the network. But if more intense computations are involved in the application a better option would be to use Java Serialization as it may reduce processing time. To derive its benefits, the serialization scheme that best fits the application is to be used.

C. *Memory Management*

By default Spark will leave 60% of space for RDD storage, 20% for shuffle memory, and the remaining 20% for user programs. Some parameters that can be used to configure the memory usage are spark. memory.

fraction, spark. storage. memory etc. Finding the best combination of these parameters would require to consider the application's object sizes, shuffle frequency etc.

D. *Hardware Provisioning*

The hardware resources that are given to Spark will have a significant effect on the completion time of your application. The main parameters that affect cluster sizing are the amount of memory given to each executor, the number of cores for each executor, the total number of executors etc.

CONCLUSION

Spark Streaming has proven to be a great tool for stream processing by providing real time analytics with minimum latency. It achieves better fault tolerance properties and scalability. It also effectively handles stragglers. Due to its unique RDD approach, it stays well ahead of the other distributed stream processing frameworks and is well adopted in the industry. Spark Streaming currently fulfills most of the requirements of real time stream processing. Future versions of Spark will further excel on those features and more to it.

Further studies can be conducted by comparing the relevant features of Spark Streaming with that of the other available distributed stream processing frameworks to better understand their capabilities and to choose the solution that would best fit the requirements of the stream processing application.

REFERENCES

[1] M. Zaharia, M. Chowdhury, T. Das, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing", Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. 2012, pp. 2-2.

[2] M. Zaharia, T. Das, H. Li. "Discretized streams: an efficient and faulttolerant model for stream processing on large clusters", Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing, 2012,pp. 10-10.

[3] M.Stonebraker, U. Çetintemel, S. Zdonik, "The 8 Requirements of Real-Time Stream Processing", ACMSIGMOD Newslett., 2005, pp. 42-47. [4] Spark Streaming Programming Guide [Online]. Available: http://spark.apache.org/docs/latest/streaming-programm ing guide.html [5] Documentation [Online]. Available: https://github.com/datastax/spark-cassandra-connector/ blob /master/doc/0_quick_start.md