Shah Zeb Mian

Comparative Analysis of Data Stream Processing Systems

Master's Thesis in Information Technology

February 23, 2020

University of Jyväskylä

Faculty of Information Technology

Author: Shah Zeb Mian

Contact information: shahzeb_2012@yahoo.com

Supervisors: Oleksiy Khriyenko, and Vagan Terziyan

Title: Comparative Analysis of Data Stream Processing Systems

Työn nimi: Vertaileva analyysi Data Stream-käsittelyjärjestelmistä

Project: Master's Thesis

Study line: All study lines

Page count: 48+0

Abstract: Big data processing systems are evolving to be more stream oriented where data is processed continuously by processing it as soon as it arrives. Earlier data was often stored in a database, a file system or other form of data storage system. Applications would query the data as needed. Stram processing is the processing of data in motion. It works on continuous data retrieved from different resources. Instead of periodically collecting huge static data, streaming frameworks process data as soon as it becomes available, hence reducing latency. This thesis aims to conduct a comparative analysis of different streaming processors based on selected features. Research focuses on Apache Samza, Apache Flink, Apache Storm and Apache Spark Structured Streaming. Also, this thesis explains Apache Kafka which is a log-based data storage widely used in streaming frameworks.

Keywords: Big Data, Stream Processing,Batch Processing,Streaming Engines, Apache Kafka, Apache Samza

Suomenkielinen tiivistelmä: Big data-käsittelyjärjestelmät ovat tällä hetkellä kehittymässä stream-orientoituneiksi, eli data käsitellään heti saapuessaan. Perinteisemmin data säilöttiin tietokantaan, tiedostopohjaisesti tai muuhun tiedonsäilytysjärjestelmään, ja applikaatiot hakivat datan tarvittaessa. Stream-pohjainen järjestelmä käsittelee liikkuvaa dataa, jatkuva-aikaista dataa useasta lähteestä.

Sen sijaan, että haetaan ajoittain dataa, stream-pohjaiset frameworkit pystyvät käsittelemään

dataa heti kun se on saatavilla, täten vähentäen viivettä. Tässä tutkielmassa tehdään komparatiivinen analyysi eri stream-pohjaisten frameworkien välillä, perustuen valittuihin ominaisuuksiin. Tutkittavat frameworkit ovat Apache Samza, Apache Flink, Apache Storm ja Apache Spark Structured Streaming. Tutkielmassa perehdytään myös Apache Kafkaan, joka on lokiperusteinen tietovarasto, jota laajalti käytetään stream-pohjaisissa frameworkeissa.

Avainsanat: Big data, Stream-käsittely, Batch-prosessointi, Streamausmoottorit, Apache Kafka, Apache Samza

List of Figures

Figure 1. log (“Apache Kafka” 2016) 6
Figure 2. Log structured storage 7
Figure 3. Kafka (“Apache Kafka” 2016) 8
Figure 4. Log anatomy (“Apache Kafka” 2016) 11
Figure 5. kafka partitions and brokers (“Kafka in a nutshell” 2020) 11
Figure 6. Kafka Log compaction (“Apache Kafka” 2016) 16
Figure 7. Samza partion (“Samza Official” 2019) 23
Figure 8. Samza job (“Samza Official” 2019) 23
Figure 9. Samza tasks (“Samza Official” 2019) 24
Figure 10. Samza Container (“Samza Official” 2019) 24
Figure 11. Storm topology (“Storm Official” 2019) 26
Figure 12. Fllink dataflow model (“Flink programming-model” 2019) 27
Figure 13. Flink parallel data flow (“Flink programming-model” 2019) 27
Figure 14. Spark Structured Streaming (“Structured streaming programming-model” 2019) 28
Figure 15. Spark Structured Streaming Programming Model (“Structured streaming programming-model” 2019) 29

Contents

1 INTRODUCTION ... 1

2 BATCH PROCESSING ... 3
- 2.1 Hadoop ... 3
- 2.2 MapReduce ... 4
- 2.3 Hadoop Distributed File System (HDFS) ... 4

3 LOGS AND APACHE KAFKA ... 6
- 3.1 Logs ... 6
- 3.2 Logs usages in different systems ... 6
  - 3.2.1 Write-Ahead Log in databases ... 6
  - 3.2.2 Log-Structured Storage ... 7
  - 3.2.3 Database replication ... 7
- 3.3 Apache Kafka ... 8
  - 3.3.1 Architecture ... 9
  - 3.3.2 Replication in Kafka ... 13
  - 3.3.3 Compaction in Kafka ... 15
  - 3.3.4 Kafka use-cases ... 17

4 STREAM PROCESSING ... 19
- 4.1 Key Features of streaming engines ... 19
  - 4.1.1 Windowing ... 20
  - 4.1.2 State management ... 20
  - 4.1.3 Message Processing Guarantee ... 21
  - 4.1.4 Fault tolerance and Recovery ... 21
- 4.2 Apache Samza - streaming ... 22
- 4.3 Apache Storm - streaming ... 25
- 4.4 Apache Flink ... 26
- 4.5 Apache Spark Structured Streaming ... 27

5 COMPARATIVE ANALYSIS OF STREAMING FRAMEWORKS ... 30
- 5.1 Windowing Mechanism ... 30
- 5.2 state management ... 31
- 5.3 processing guarantee ... 32
- 5.4 Fault tolerance ... 34
- 5.5 Other Comparisons ... 35

6 CONCLUSION ... 38

BIBLIOGRAPHY ... 39

1 Introduction

Recent years have witnessed an exponential increase in magnitude and velocity of data. In the 20th century, most of the data on the internet was in the form of transactions e.g sales data on a shopping website, banking transactions, etc. Relational databases were used to store and process the data. However, At the start of the 21st century, organizations have started generating new kinds of data e.g activity-based data on websites, click-stream data, and sensors data. To process these new kinds of data, Organizations are using batch processes. For example, a business may run a batch process by the end of the month to calculate bills of each of their customer or an IT company might run a job to analyze the system logs on hourly basis. They are very successful for these use-cases. But they can not be used for a use-case where the data needs to be processed in near real-time. For example, showing a user some recommendations based on the recent activity. The reason is that in batch processing first, the data needs to be collected in batches and then wait for the whole data to be processed. Certain types of data streams such as stock values, credit card transactions, traffic conditions need to be processed in near real-time to get useful insights from it. Traditional processing systems have shortcomings for processing data in a near real-time fashion. Therefore, to process the data in near real-time streaming frameworks were developed. Stream processing applications are long-running processes that continuously consume one or more streams of data, process them and produce results. The result can be output to a front end or saved to a database or can be input to another process.

In the thesis, we will study these streaming frameworks and at the end do a comparative analysis of selected streaming frameworks namely Apache Samza, Apache Flink, Apache Storm, and Apache Spark Structured Streaming. The comparison will be based on the features which are essential to streaming frameworks.

The rest of the thesis is organized as follows. Chapter 2 focuses on batch processing and explains certain concepts related to batch processing. Chapter 3 Introduces the Apache Kafka and explains Apache Kafka in detail. Chapter 4 introduces Streaming. It explains streaming in detail, and selected streaming engines are introduced. Chapter 4 also explains the essential features of a streaming engine. In chapter 5, a comparison has been made between the

selected streaming engines based on the features of a streaming engine. Finally, Chapter 6 concludes the work.

2 Batch Processing

Batch processing frameworks process Big data across the cluster of computers. A batch processing job takes a large amount of data, process it and produce output results. Batch processing frameworks can be scaled to thousands of computers. (“Hadoop Official Website” 2020). A batch job takes some time to process data. Time can be from a few minutes to several days. A batch processing system works on bounded data. Bounded data is a finite set of data. In bounded data, we already know that data is completed and we are not expecting new data to arrive. An Example of bounded data can be a daily sales of a website, hourly GPS data generated by a GPS sensor. Batch processing is an efficient way of processing huge volumes of data. Data is collected in batches, processed with a batch job and results are produced.

2.1 Hadoop

Apache Hadoop is an open-source framework for distributed storage and processing of a large amount of data in a scalable way. Hadoop can be scaled to thousands of computers. Apache Hadoop stack consists of the following modules.

HDFS

YARN

MapReduce

Common Libraries

Hadoop uses HDFS for storage purposes and MapReduce for processing of data. HDFS breaks data into small chunks and distributes across the nodes in the cluster. Apache Hadoop uses YARN for job scheduling and resource management of clusters. Common libraries supports different Hadoop modules.

2.2 MapReduce

MapReduce is a programming paradigm for processing large amount of data. MapReduce is the heart of Hadoop[1]. The applications can be run in parallel on clusters (thousands of nodes) in a reliable and fault-tolerant manner. The system automatically distributes the workload over different nodes in cluster. Typically MapReduce uses a file-system for data storage (HDFS). The MapReduce system itself takes care of scheduling tasks, monitoring, and re-execution of failed tasks. According to Google (Dean and Ghemawat 2008) , they have used MapReduce in many projects. MapReduce become popular due to several reasons. First, MapReduce model is very simple to use, even for the programmers without having experience of parallel and distributed computing. As the framework itself handles parallelization,fault-tolerance, monitoring, and load balancing, etc. Second, MapReduce can be scaled to large cluster of machines comprised of thousands of machines. The implementation of MapReduce makes it easy to use the resources of thousands of machines. Finally, Large number of problems can be easily expressed as MapReduce computations e.g, filtering, counting and sorting. (Dean and Ghemawat 2008)

The programming model of MapReduce is based on MapReduce Functions. A user uses two MapReduce functions: Map and Reduce. Map function takes an input and produces an intermediate result in the form of key/value pairs. The MapReduce library collects all the key/value pairs produced by the map function, group all the values having same key and passes it to reducer function. Reduce function gets a key and all the values associated with that key and merges the values into a single output. (Dean and Ghemawat 2008)

2.3 Hadoop Distributed File System (HDFS)

HDFS is a distributed file system designed to store a large amount of data in a reliable fashion. HDFS runs on commodity hardware. HDFS provides high throughput of data and is an excellent tool for storing a large amount of data. HDFS is highly fault-tolerant. (“HDFS Design” 2020). HDFS is designed for batch processing. Hadoop application have large data

1. The Apache™ Hadoop® project develops open-source software for reliable, scalable, distributed computing. https://www-01.ibm.com/software/data/infosphere/hadoop/mapreduce/

sets, usually from gigabytes to terabytes and HDFS takes care of these large data sets. HDFS is designed in a way that can scale to hundreds of nodes in a cluster.

HDFS architecture is based on master/slave paradigm. HDFS has a single master node called NameNode and other dataNodes. Data Nodes are usually one node per cluster. NameNode keeps the metadata about the data on each dataNode and keeps track of all the files. Client's applications use NameNode to access the files for reading or writing data. DataNodes manage the storage attached to nodes. HDFS stores user data in files by exposing the file system namespace. A file in HDFS is split into blocks and blocks are stored on DataNodes. The DataNodes takes care of read/write request from the client's file system. The DataNodes also takes care of block creation, deletion and replication under the supervision of NameNode. The NameNode and DataNode are software that runs on commodity machines. HDFS is written in Java and any machine that supports Java can run it. Because Java is highly portable and can be run on different machines, HDFS can be deployed on different types of machines. A typical deployment of HDFS has a dedicated machine that runs the NameNode and other machines that runs a single instance of DataNode. Together they form a cluster of HDFS.("HDFS Design" 2020). Having a single NameNode in a cluster simplifies the architecture but it also means that NameNode is a single point of failure in HDFS. ("NameNode - HADOOP2" 2020)

HDFS supports a traditional hierarchical file organization. A user or an application creates directories and files are saved in these directories. For fault tolerance, HDFS replicates the files by replicating the blocks of a file. An application can choose the replication factor by itself. HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file. The NameNode takes care of the replication of blocks. It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster. Receiving a heartbeat means that DataNode is functioning correctly while Blockreport is a list of all blocks on that particular DatNode.

3 Logs and Apache Kafka

This chapter explains Logs, their usage in different systems and Apache Kafka.

3.1 Logs

A Log is an append-only, sequence of records ordered by time. According to Jay 2013, Log is perhaps the simplest storage abstraction. Records are appended to the end of the log. Reads proceed from left to right. Each record that appends to log, gets a unique sequential number that acts as a unique key in the log. This unique number can be thought of as the "timestamp" of the entry. Many systems use logs for different purposes. We will explain them in the next section.

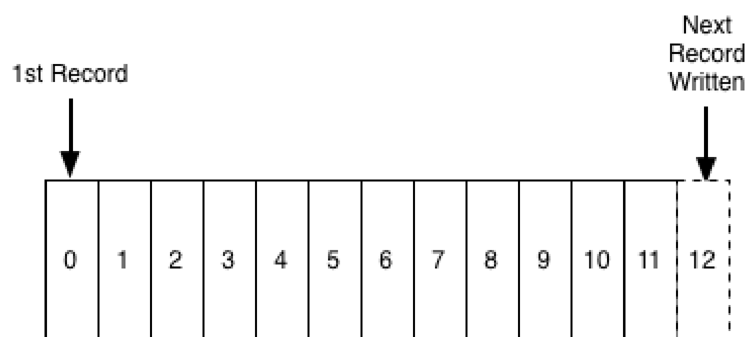

Figure 1. log (“Apache Kafka” 2016)

3.2 Logs usages in different systems

Relational databases and some other systems uses logs internally for different purposes.

3.2.1 Write-Ahead Log in databases

Relational databases e.g MySQL use write-ahead logs to make it reliable and consistent.(Kleppmann 2016b). A write-ahead log (WAL) in the database is an append-only file. Before making any actual changes to the database state, the changes are first written to WAL and saved it to disk.

After saving the changes to disk, storing-engine makes the actual changes to the database. This makes the database reliable and consistent. If a database crashes, WAL can be used to recover it and bring it back into a consistent state.

3.2.2 Log-Structured Storage

Some storage engines like HBase and Cassandra use the log as a primary storage medium, known as log-structured storage. (Kleppmann 2016b). In log-structured storage, data is not always appended to one single file because a single log file will become very large and finding data in that file will become too difficult. Therefore the log is broken into segments. The segments in log-structured storage are merged from time to time and duplicate keys are discarded as shown in the figure.

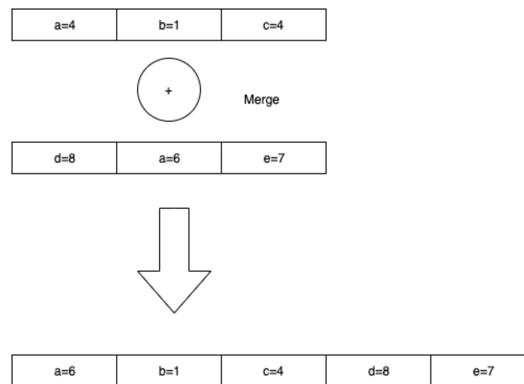

Figure 2. Log structured storage

3.2.3 Database replication

Relational databases use logs for replicating data between the databases. As discussed earlier that databases use the log to keep in sync the data-structures and indexes of database hence it can also be used to keep remote replica databases in sync with the master database. Oracle, MySQL, and PostgreSQL include log shipping protocols to transmit portions of the log to replica databases that act as slave databases. Oracle has two products XStreams and GoldenGate which use logs for sending changes of databases to client applications ("Oracle Xstream" 2020) ("Oracle goldengate" 2020). similarly, MySQL and PostgreSQL also

use logs for replication of databases (Jay 2013). PostgreSQL uses the WAL to replicate the database while MySQL uses a separate replication log for replicating the database.

Data systems use logs over the years for different internal implementation. In the next section, I will discuss Apache Kafka which is also built around the logs but rather than using the logs as an internal implementation, Kafka exposes the logs to users and users build applications around it.

3.3 Apache Kafka

Apache Kafka[1] is a scalable, real-time publish-subscribe messaging system rethought as a distributed commit log. It incorporates the idea of messaging systems and logs. Kafka is a distributed, partitioned, replicated commit log service. According to (Wang et al. 2015) Kafka was built by LinkedIn Engineers. According to LinkedIn Kafka handles more than 10 billion messages writes per day (Jay 2013). Since Oct. 2012, Kafka has become a top-level Apache open-source project (Jay 2013). The motivation behind building Kafka was to make a data pipeline system that can handle real-time data feeds by ingesting them from different producers and deliver it to different consumers.

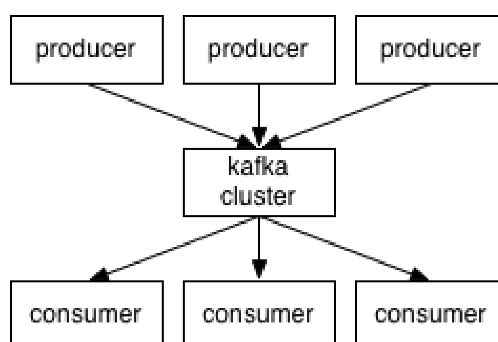

Figure 3. Kafka ("Apache Kafka" 2016)

.

Apache Kafka is designed having in mind the following characteristics. (Garg 2015)

1. http://kafka.apache.org/

- **Persistent storage**

  Kafka stores every message persistently on disk and is replicated across multiple disks to prevent data loss. Kafka is designed with O(1) disk structures, so even with a very large volume of data, it gives a constant performance.

- **High Throughput**

  Keeping in mind a large amount of data, Kafka is designed to handle hundreds of MBs of reads and writes per second from large numbers of producers and consumers

- **Distributed**

  Kafka is distributed in nature. It supports message partitioning over Kafka servers and distributing consumption over a cluster of consumer machines while maintaining per-partition ordering semantics. Kafka cluster can grow elastically.

- **Real-time**

  Kafka system supports real-time availability of data. It immediately provides Data produced by producers to consumers.

At a higher level, Kafka is a data pipeline in which data is arriving from different data sources. These sources can be called producers or publishers. Kafka then serves this data to different subscribers or consumers. ("Apache Kafka" 2016)

3.3.1 Architecture

Before explaining Kafka architecture first I will explain some key concepts of Kafka:topics, producers, consumers, and brokers

Topics

All Kafka messages are organized into Kafka topics. A Topic is the stream of messages of a particular type. A Producer publishes messages to a topic. A Consumer consumes messages from the topic. For Scalability purpose A Topic is divided into multiple partitions. Each partition is an ordered, immutable, writes-ahead log of messages. Each message in a partition is assigned with a unique id called offset. Offset uniquely identify each message within a partition. A partition can be replicated across

the number of servers for fault tolerance. Users choose the replicated factor. One of the partitions in these replicas becomes leader while others become followers. A leader is responsible for all the read/writes operations of a given partition. If a leader fails, another replica takes over as a new leader. Topics are saved on a set of servers called brokers.

Producers

Producers publish data to Kafka Topics. As Topic is divided into multiple partitions, the producer chooses which message will be assigned to which partition within the topic. If a key is provided, then messages having the same key will be assigned to the same partition. If the key is not provided, then messages will be assigned to partitions in a round-robin fashion.

Brokers

Since Kafka is distributed in nature, a Kafka cluster typically consists of multiple servers called brokers. Each Topic is persistently saved on these brokers. Each broker store one or more partition of a particular topic.

Consumers

A consumer is subscribed to one or more topics and consume messages to process. A consumer can be an Off-line system like Hadoop or traditional data warehouse system or it can be a NoSQL data store such as HBase for near real-time analytics. Similarly, it can be a streaming engine like Apache Samza or Apache Storm for real-time processing of streams. In Kafka, consumers are in the form of groups. A Kafka consumer consumes messages from a unique partition of a topic. This means that no two consumers of the same group can consume messages from a single partition of a topic.("Apache Kafka" 2016)

In Kafka the key abstraction is Topic. As discussed above, a topic is a stream of messages of a particular type. For example, the temperature data collected from a weather station can be a topic. A Producer publishes messages to the topic. Topics are divided into a number of partitions. Messages are stored persistently on the Kafka broker. Each time producers send a message, it is appended to one of the partitions. Hence each partition is a logical log. Physically each log is implemented as a set of segment files of approximately the same size. Users can selects the number of messages accumulated before flushed to disk or set the time

interval to flushed the messages to disk. Messages are available to the consumers only after it has flushed to disk. Each message in a partition is assigned with a unique id called offset which uniquely identifies the message within the partition. For fault tolerance, each partition is replicated across the number of brokers. Each partition has one broker which serves as a leader and zero or more brokers which serves as followers. All read and write operations are handled by the leader and followers follow the leader. If a leader fails then Kafka elects a new leader. To balance the load within the cluster, each Kafka broker acts as a leader for some partitions and followers for other partitions.("Apache Kafka" 2016)

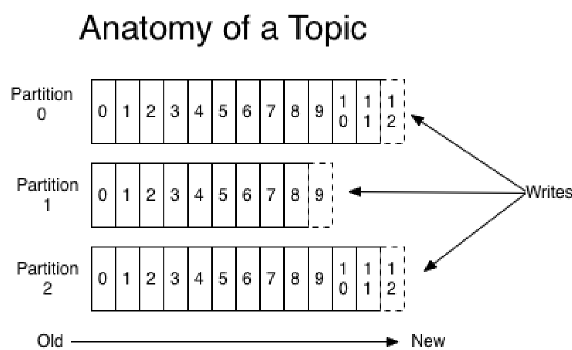

Figure 4. Log anatomy ("Apache Kafka" 2016)

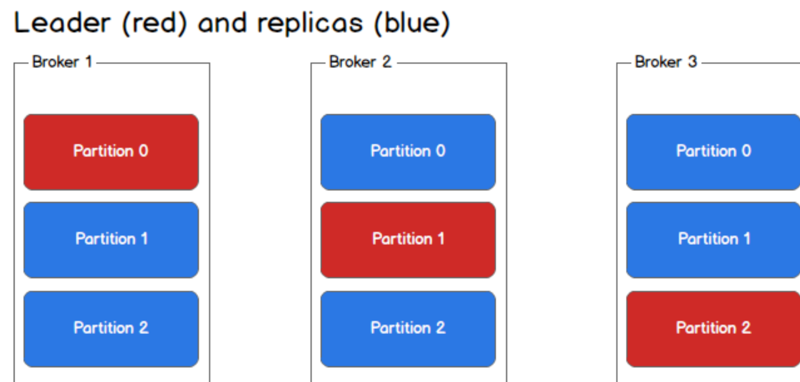

Figure 5. kafka partitions and brokers ("Kafka in a nutshell" 2020)

A consumer subscribes to one or more topics by creating message streams for that particular topic. A consumer can then consume messages from the topic. Kafka uses "pull" model to deliver messages to consumers. The info about how much messages are consumed by consumers are tracked by the consumer itself. Consumers do it by consuming messages

sequentially from a particular partition and acknowledges the offset id, which means that the consumer has received all the messages prior to that id. The consumer sends an asynchronous pull request to the broker to have data ready for application to consume. Each pull request contains the offset of the message from which consumption of messages begins and the number of bytes of data to consume. As the state about messages consumption is maintained by consumers, it has one great advantage that if there is some error in programming logic of consumer or data is lost due to the crash, a consumer can rewind back to older offset and can re-consume the data. Kafka has the concept of consumer groups. Each consumer group has one or more consumers. Messages from a single partition within a topic are consumed by a single consumer in the same consumer group. This means that the same partition can not be consumed by two consumers in the same group. There is no coordination required between the two consumers about who consumes what messages and hence no state management is needed between different consumers.

Kafka decouples data pipelines. The publishers and consumers do not need to know each other. A performance issue on consumers do not impact producers. Similarly, different consumers do not have any impact on each other. Kafka uses Apache Zookeeper for different coordination needs.

At a high level, Kafka gives the following guarantees about messages.

- Message sent by a producer to a particular topic will be appended in the order they are sent. For example, If two messages M1 and M2 are sent by the same producer to a particular topic partition and M1 is sent first then M1 will have a lower offset than M2 and will appear earlier in the log.
- A consumer consumes messages in the order they are stored in the log.
- For a topic with replication factor N, Kafka tolerates up to N-1 server failures without losing any messages committed to the log. (more explained in subsection Replication in Kafka below).("Apache Kafka" 2016)

3.3.2 Replication in Kafka

Replication is keeping a copy of the same data on multiple machines that are connected through a network (Kleppmann 2016a). There are several reasons data is replicated.

- It reduces latency by keeping data geographically closer to users.
- It increases the availability of the data. The system can continue to work if there is a failure in some part of the system.
- It increases read throughput by increasing the number of servers that serve read requests.

A Kafka topic is replicated across a number of brokers. Kafka guarantees that a message will not be lost and will be available for consumption even when there are failures. Failure can be machine error, program error, etc. Kafka allows automatic failover [2]. If a server in the cluster fails, messages will be still remain available to consume. Each partition in Kafka has a single leader and zero or more followers. Leader and followers constitute the replication factor in Kafka. Normally, there are more partitions than brokers. Leaders are evenly distributed among the brokers for load balancing. All read and writes operation goes through the Leader. Follower consumes messages from the leader the same way normal consumer consume from them. The replicas which are alive and caught-up to the leader are called in-sync replicas (ISR). Zookeeper keeps track of in-sync replicas. If a follower dies or lags from the leader, the leader removes it from the list of in-sync replicas. A message is considered to be committed when all in-sync replicas get it. Messages are available for a consumer to consume only after they are committed. The guarantee that Kafka offers is that a committed message will not be lost, as long as there is at least one in-sync replica alive. If a leader dies, Kafka handles it by electing a new leader from the in-sync replicas. Zookeeper maintains a list of all in-sync replicas and any of the in-sync replica can ba a contender of becoming a leader. There are two common strategies to keep replicas in-sync. 1. Primary backup 2. Quorum based . In Quorum based replication one replica is chosen as Leader and the rest are followers. All write requests go through the leader first and the leader then waits until the write happens in the majority of the replicas. Only then the write is committed and

2. if a leader fails, system elects new leader, all reads and writes go to new leader and followers now follow the new leader. This process is called failover

available to consume. The size of replicas does not change even if some replicas are down. If we have $2f+1$ replicas, quorum based replication can tolerate up to f failures. In the case of leader failures, quorum based replication needs at least $f+1$ replicas to elect a new leader. In primary back-up replication, the leader waits until the write happens on each replica. If one of the followers is down, Leader removes it from the replica's list and continuous to write to remaining replicas. If a failed replica comes back it is allowed to rejoin the group. Primary back-up with f replicas can tolerate $f-1$ replicas (Neha 2013).

The tradeoffs between two approaches are as follows:

1. Quorum based replication has better write latency than the Primary back-up because we do not need to wait for the write to happen on all replicas.
2. With the same number of replicas, Primary back-up can tolerate more failures. (Neha 2013).

According to Kafka (Neha 2013), they have chosen the primary-backup replication in Kafka since it tolerates more failures.

It is also possible that all replicas die. Two approaches can be made in this situation.

- Wait for an in-sync replica to come back and that replica can be chosen as leader
- Choose a non-in-sync replica as leader

This is a trade-off between consistency and availability. If we wait for an in-sync replica there might be a chance that it has gone down permanently while if we take a non-in-sync replica as a leader then there might be a chance that it has not all the data. By default Kafka chooses second option. A non-in-sync replica becomes leader although it is not guaranteed to be consistent. This option can be disabled using configuration where downtime is preferable to inconsistency.

Availability and Durability Guarantees

Writing in Kafka can be done in a synchronous or asynchronous fashion. In asynchronous replication, the leader acknowledges the clients about the commit as soon as it finishes writing to its own local log. The Leader does not wait for the followers to

catch up it. There is no guarantee that a commit message can survive any failure. In synchronous replication, the leader first writes the message to its local log. Followers then pull the message from the leader and write it to their own logs. Leader waits for followers to commit message in their logs. Followers send an acknowledgment to the leader and the leader sends the acknowledgment to the client that write has been committed. The acknowledgment is only from in-sync replicas. If we choose asynchronous replication then durability would be a problem, because if a leader die and another leader is chosen among the followers there could be a possibility that some messages will be lost. On the other hand, if synchronous replication is chosen then availability would be a problem but there is another option that can be used with synchronous replication. Kafka gives an option to set a minimum ISR size. We can specify a minimum ISR size by ourselves. In this case, a write will only be acknowledged if the size of the in-sync replicas is above the minimum ISR size. This offers a trade-off between consistency and availability. A higher value for minimum ISR guarantees better consistency but it reduces the availability since the partition will be unavailable for writes if the number of in-sync replicas drops below the minimum value. ("Kafka Official" 2016).

3.3.3 Compaction in Kafka

As discussed earlier Kafka uses persisted logs for storage. Having data in the form of logs is very useful because we can replay them and do our computation whenever we want. However, the problem with logs is that they are ever-growing and the disk can go out of space at some time. Therefore, Kafka deletes older records from logs to free up space. Originally Kafka had only supported the time window retaining mechanism: deleting old messages from the log after some fixed time. By default, it is configured for seven days but users can set it according to their requirements. It works well for temporal systems. For example, A website owner might be interested to count the number of website visitors per day. Users can set the window time of 24 hours and messages will be retained only for 24 hours. Similarly, if the log reaches some predetermined size then messages are deleted. But it would be a problem in some use-cases to delete the data by time -based retention: if a Kafka log

is used to store the shipping addresses of customers. In that case, retaining messages by time and size will not work and the system needs to store the last shipping address for every customer. Another example is, Apache Samza (discussed in chapter 4) uses Kafka to store its current state of an application. Every time the state changes, Samza writes the new state into the Kafka log. In case of a crash, Samza can read Kafka and recover its latest state. In this case, Kafka needs to have only the latest state every time. In these cases, data can not be deleted simply by time-based approach therefore Log compaction adds an alternative retention mechanism, retaining messages by key instead of purely by time. For every unique key Kafka keeps the last value and throws away the duplicate values. This means Kafka will have a recent update for each key in the log. Log compaction addresses use cases such as restoring state after application crashes, system failure or reloading caches after application restarts during operational maintenance. This retention policy can be set per-topic. A Kafka cluster can have some topics where retention is by size or time and other topics where retention is by compaction(key-based).

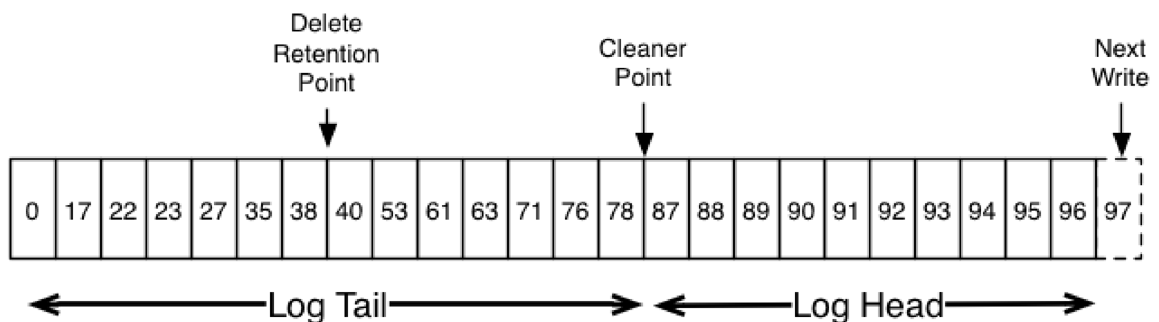

Figure 6. Kafka Log compaction (“Apache Kafka” 2016)

The figure shows a logical structure of Kafka log with offset of messages. The head of the log is more like a traditional Kafka log where offsets are sequential ids. The Tail of the log shows compaction in Kafka. Compaction ensures that each key is unique at the tail of Log. Every Key in the tail has only the latest value. Each message in the tail still has original offset assigned when they were first written. In the figure, 24,25,26,27 and 28 are all equivalent positions. A read beginning at any of these offsets would return a message set beginning with 28.

Compaction is also used for deletes. A message with a key and null value will be treated as a delete from the log. Kafka keeps this message for a configurable amount of time. During this time consumer will see this message and know that that the value is null. The deleted messages are no longer retained after "delete retention point".

Log compaction is handled by a pool of background threads that recopy log segment files, removing records whose key appears in the head of the log. Cleaning does not block read/write operations and logs serve the requests normally. Log compaction guarantees the following:

- Ordering of messages is always maintained.
- The messages will have sequential offsets and the offset never changes.
- Reads progressing from offset 0, or the consumer progressing from the start of the log, will see at least the final state of all records in the order they were written.

A pool of background threads called log pool handles log compaction operation.it works as follows.

- It chooses the log that has the highest ratio of log head to log tail.
- It creates a summary of the last offset for each key in the head of the log.
- It recopies the log from beginning to end removing keys which have a later occurrence in the log

(Jay 2016),(“Kafka Introduction” 2016)

3.3.4 Kafka use-cases

Kafka is used as Messaging, Log aggregation, Event sourcing, Stream processing and commit logs.

Kafka provides better throughput, built-in replication, and fault-tolerance, therefore it can be a good fit in applications, where large scale message processing is needed.

Typically Log aggregation collects logs files from the server and puts them in a central file e.g HDFS. Kafka can be used to collect the log data as a stream of messages from multiple sources and consumers can then consume the data from Kafka message’s stream and process

it in low latency fashion.

Event sourcing is an application type where changes are logged as a time-ordered sequence of records. Because Kafka can store very large data in totally ordered style, it makes Kafka a good back-end type for an event sourcing application

In a distributed system, the Kafka commit log helps replicate data between nodes and works as a re-sync mechanism for restoring data on failed nodes.

Stream processors use Kafka for providing different streaming functionality to users. A Stream processor consumes Kafka topics, process it and transform into new Kafka topics. Stream processors use Kafka for different operations like aggregations, filtering and joining.

Similarly, other use-cases of Kafka are website activity tracking, Metrics, IoT solutions, etc.

4 Stream Processing

Stream processing is the processing of data in motion i.e processing data as at is produced or received. Data is produced in the form of continuous streams. For example, sensor events, user activity on a website, stock exchange data, database changes. All types of data is created as a series of events over time. The ability to process data as a continuous stream is becoming an essential part of building a data-driven organization. Getting meaningful and timely insights from unbounded data is very challenging and traditional batch processing such as Hadoop has shortcomings for it. Batch processing frameworks have to wait for data to be collected in batches and then process it, therefore they produce results with latency. In many applications like fraud detection in credit cards, healthcare systems using sensors, IoT, data must be processed as it is produced. Because in certain use cases such as stock exchange prices, traffic conditions, credit card transactions if data is not processed immediately, it become less important. Unlike Traditional data systems that collect and periodically process large static data, streaming frameworks process data as it becomes available. This chapter introduces some popular streaming frameworks. It will focus on discussion of various real-time available streaming frameworks. These frameworks can store, analyze and predict results for enormously large bulks of data in considerable amount of time. Also, the important features of streaming frameworks are explained .

4.1 Key Features of streaming engines

Following are some of the Key features of a streaming engine.

1. Windowing
2. State management
3. Message processing Guarantee
4. Fault Tolerance And recovery

4.1.1 Windowing

Windowing is a concept of taking an infinite stream and splitting into chunks of finite streams. For example, a traffic sensor that counts every 15 minutes the number of vehicles passing through a certain location. Streaming engines provide these kinds of jobs by windowing. User sets a window size of 15 minutes, counts the number of vehicles, aggregates the result and discard window after 15 minutes. A window is usually defined either by time duration or record count. Count-based window is sized by the number of records included in windows. An example of count-based window can be every hundred readings from an Iot device. Time-based windows can be fixed windows, sliding windows and session windows. Fixed windows split streams into fixed-size segments, each with a fixed temporal length, start time and end time. Incoming data goes to one and only one window in fixed windows. Sliding windows have a fixed length but are separated by an interval time. If the interval time is less than the window's length than the windows overlaps and incoming data will go to more than one window. If the interval is equal to windows length then sliding windows become fixed windows. A Session window does not have a fixed time length and is composed of a series of events terminated by a gap of inactivity greater than some threshold time. Session windows do not overlap and do not have a fixed start and end time. Time notion for windowing can be event-time or processing-time. Some processing engines have only support for processing-time while others have for both event-time and processing-time: processing-time is the time at which event is processed by the streaming engine while event-time is the time at which event occurred.

4.1.2 State management

State management is one of the important features of streaming engines. Some streaming operations are stateless by nature and there is no need for maintaining their state e.g filter or map operations. But some operations are stateful and there is a need for maintaining the states for these operations. An example of a stateful operation is count e.g counting the number of visitors that visits a website every day. Streaming engines have support for maintaining states for stateful operations. Some streaming engines provide in-memory state management while other stores state locally on the same machine. Similarly, some streaming

engines maintain state remotely in a database. Some streaming engines use all of these approaches for state management.

4.1.3 Message Processing Guarantee

Message-processing guarantee is an important feature of streaming engines. The semantics for message-processing guarantee are at-most-once, at-least-once, and exactly-once. In an at-most-once semantics, the streaming engine ensures that a message will at most processed once, means that a message might be lost in the process. This is the least favorite guarantee semantics because if a message get lost during the process, no effort will be made to deliver it and hence it is the least fault-tolerant among all. In an at-least-once semantics, a streaming engine ensures that a message must be processed at least once but a message might be processed more than once in some cases. For example, if a failure occurs and the job needs to reprocess again, some messages might be processed more than once. In an exactly-once semantics, the streaming engines ensures that a message must be processed exactly once. In exactly-once processing, the message can't be lost or processed multiple times and it is the most fault-tolerant semantics. The choice of these semantics involves a trade-off between reliability and cost. for example, in some applications, faster processing having weaker guarantees may be fine but in other applications, a reliable process with a stronger message guarantee will be required. Streaming engines support message processing semantics by different techniques like snapshotting, write-ahead logs, recovery algorithm, and acking mechanism.

4.1.4 Fault tolerance and Recovery

Fault tolerance is one of the important features of streaming engines. Streaming applications are often real-time applications and reprocessing it from the start is impractical if a failure occurs. Therefore, in case of failures e.g network failure, node failure, etc, a streaming engine should be able to recover and start processing again from the point where it left. Streaming engines achieve fault tolerance through different techniques e.g changelog, checkpointing and savepoint.

other important features for a streaming engines are programming Model, data source interaction model, data partitioning strategy, deployment, community support, support for high level languages, support for storage systems and support for input sources

4.2 Apache Samza - streaming

Apache Samza [1] is a streaming engine that is used to process data in real-time. Apache Samza uses Apache Kafka for messaging, and Apache Hadoop YARN for fault tolerance, processor isolation, security, and resource management. Samza Provides a very Simple API comparable to MapReduce. Samza manages its state by keeping a snapshot of its state. Samza is extremely fault-tolerant. Whenever a node in the cluster fails, Samza works with YARN to migrate the task to another node. Samza uses Kafka for message processing and Kafka guarantees that messages are processed in the order they are written to a Kafka partition. Though Samza works out of the box with Kafka and YARN, Samza provides a pluggable API that lets you run Samza with other messaging systems and execution environments. ("Samza Official" 2019)

Apache Samza has some key concepts: Streams, Jobs, Partitions, Tasks

Streams

Samza processes data in the form of streams. A Stream is a collection of immutable messages of the same type. Examples of Streams are all the clicks on a website, temperature data send by a sensor, events send by an IoT device. A stream can have any number of consumers that can read the messages. Streams are always persisted by the Samza in its brokering layer. Messages can optionally have an associated key that is used for partitioning.

Partition

To achieve Scalability, Each stream in Samza is further divided into Partitions A Partition is a sequence of messages in an ordered form. Each message in a Partition has an identifier called offset. Offset uniquely identify each message in the partition. Offset can be of different type e.g sequential integer, byte offset, and string. When a message

1. http://samza.apache.org/

is written to a stream, it is appended to one of the partition. .

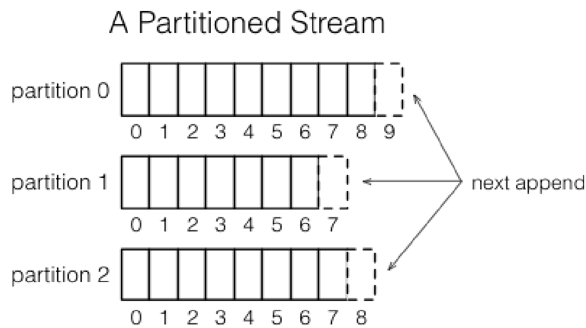

Figure 7. Samza partion ("Samza Official" 2019)

Jobs

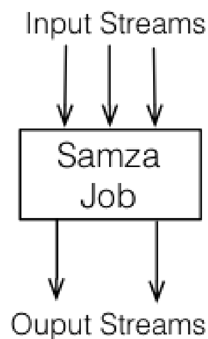

Figure 8. Samza job ("Samza Official" 2019)

A Samza Job is a code that performs a logical transformation on a set of input streams and produces output streams. For scalability purposes, A job is further divided into tasks.

Tasks

Like Samza streams A job is further divided into tasks for achieving scalability. Each task consumes data from one partition for each of the job's input streams. Messages are processed sequentially from the input partition. As ordering is provided within the partition and there is no ordering across the partitions, each task consumes from one partition of the same input streams.

YARN takes care of the assignments of tasks to each machine. A job can be distributed across many machines

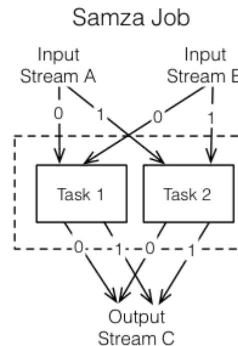

Figure 9. Samza tasks ("Samza Official" 2019)

Containers

Both Tasks and partitions are the logical units of parallelisim in Samza. A container is the physical unit of parallelism. A container is Unix or Linux process. A container runs one or more tasks. The number of tasks is determined by the number of partitions. The number of containers (and resources with it) is determined by the user at run time and can be changed. .

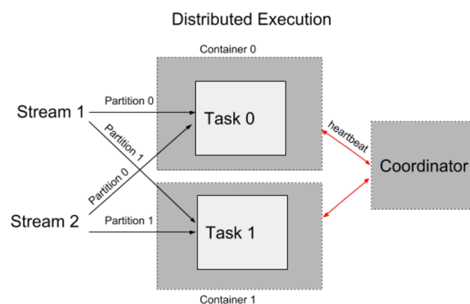

Figure 10. Samza Container ("Samza Official" 2019)

Together these all make a Samza application. A Samza application processes messages from input streams, transform them and save results to an output stream or a database. It chains multiple operators, each of which transforms one or more input streams. ("Samza Official" 2019)

4.3 Apache Storm - streaming

Apache Storm is one of open source, real-time streaming framework originally created by Backtype and later acquired by Twitter (Toshniwal et al. 2014). It is written in Java and Clojure. Apache Storm is an open-source distributed framework that is known for its low latency, reliability and fault tolerance. It follows in-memory computation mode (Dendane et al. 2019). Storm braces many use cases like online machine learning, real-time analytics, extract transformation load (ETL) and continuous computation (Wingerath et al. 2016). Storm architecture is based on a master-slave paradigm. The cluster is made up of one main node and many other slave nodes. The main node in storm architecture is known to be "Nimbus". Nimbus allocates codes, distributes data among worker nodes, assign tasks and monitor errors. Nimbus is similar to Hadoop's Job tracker in functionality (Alkatheri, Abbas, and Siddiqui 2019). Slave node is known to be the supervisor node that follows the instructions given by the nimbus. It further delegates tasks to the work processes. Work processes further hand-over tasks related to a specific topology to Executors. Single work process can have multiple executors. The task is the actual data transaction performed. It is either a Spout or Bolt.

The three main concepts in Apache Storm are spout, bolt, and topology. A spout is a source of stream in Apache storm. Usually, A spout reads from message queue like Apache Kafka but a spout can generate its own streams also. Similarly, a spout can read from other sources like Twitter stream API. A bolt processes the input streams and produces output streams. A topology is the network of spouts and bolts. Storm is none different than a road with several connecting checkpoints. Data traffic begins at a point and passes through various checkpoints in topology. Graph of computation is topology in Storm and is treated as directed acylic graph. Traffic in the topology is data stream that enters through a retrieving point that is spout. Spout is like a stimuli starting the journey, data passes through receiving terminals called bolts. In bolts, the data is filtered, sanitized, aggregated and analyzed . Data from spout would flow to one or multiple bolts. Bolts further may forward received data to another bolt or output it to UI. . Nimbus and supervisor nodes interact with one another via Zookeeper. It is a service that is used for robust and synchronized data sharing. States of both; nimbus and supervisor are maintained by Zookeeper. Nimbus that fails to share data

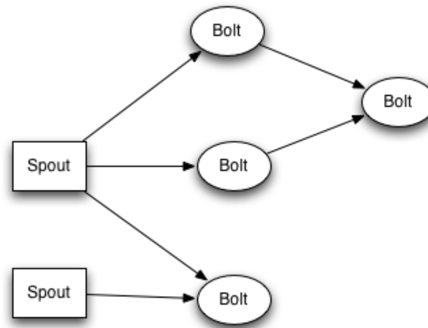

Figure 11. Storm topology ("Storm Official" 2019)

can restart its state right from where it stopped working. ("Storm Official" 2019)

4.4 Apache Flink

Apache Flink was started under project Stratosphere and is considered to be a top-level Apache project built on a philosophy that entertains different classes of data processing applications. Batch data computation, real-time systems analytic, continuous data streaming and machine learning analysis are key computations provided by Flink (Carbone et al. 2015). For reading and writing data, Flink can use different systems like HDFS(Hadoop Distributed File system), S3(Simple Storage Service from Amazon), RDBMS(Any Relational Database), Hbase(NoSql Database), Flume (Data Collection and Aggregation Tool), KAFKA(Distributing Messaging Queue) and MONGODB(NoSql Database). Data in Flink can be processed both in bounded and unbounded fashion. Flink can be run standalone, on YARN, on Mesos and on AWS. Flink also embeds master-slave prototype. Manager/ Master node receives tasks from the client further, it distributes and submits it to slave nodes. Distributed computing enables Flink to compute data efficiently and rapidly. Slave node in cluster happens to be task manager executing work assigned by Job manager that happens to be a centralized master node. A noteworthy attribute of Flink architecture is accurate data ingestion. It continues data processing at high throughput with low latency. Kernel/ Core layer in Flink is a distributed streaming engine that executes data flow programs. The building block of Flink application is streams and transformation. A stream is a flow of data records and transformation is an operation that takes one or more streams as input and produces one

or more output streams as an output result. The programs are represented as DAG (directed acyclic graphs). . To achieve parallelism A stream in Flink is further divided into partitions

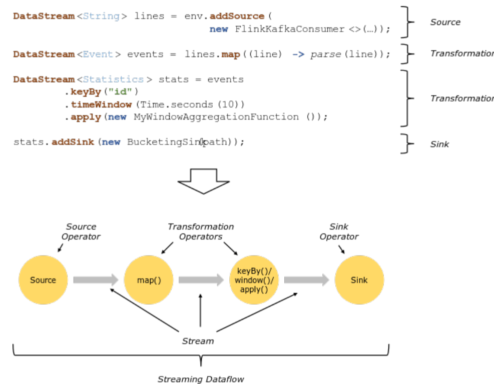

Figure 12. Fllink dataflow model ("Flink programming-model" 2019)

and each task (operator) is subdivided into operator subtasks. The operator subtasks are independent of each other and can be executed on different machines or containers. operator's parallelism is the number of operators subtasks while the parallelism of streams is always the number of subtasks operator of the producing operator. ("Flink programming-model" 2019)

.

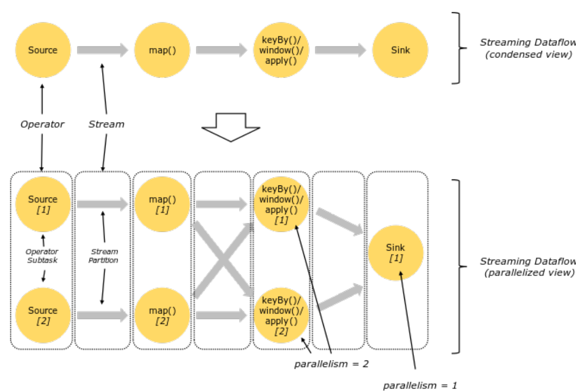

Figure 13. Flink parallel data flow ("Flink programming-model" 2019)

4.5 Apache Spark Structured Streaming

Structured Streaming is a streaming engine built on top of the Spark SQL engine. It can be run as standalone, on EC2, on Hadoop YARN, etc. Structured streaming reads data

from an input source and write it to output source called sink. Input sources can be file source, Apache Kafka, Apache kinesis and SQL databases. Output sink can be apache Kafka, Hadoop, SQL databases, etc. Structured Streaming programming model is based on continuous processing. The input data stream is considered as a table where data is continuously appended and spark runs an incremental query on the data stream. Every time new data arrives, it is appended to the input table . Querying input table will generate "Result

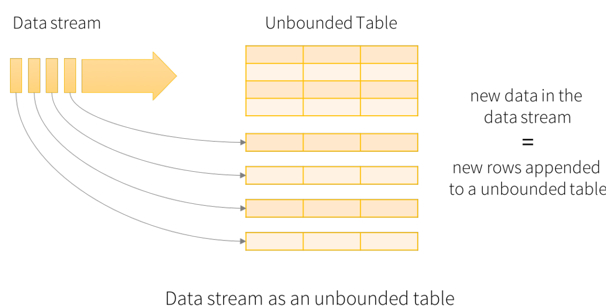

Figure 14. Spark Structured Streaming (“Structured streaming programming-model” 2019)

Table". On every trigger interval, new rows get appended to the input table and update the Result Table. Triggers control how often Spark will compute a new result and update the Result Table. whenever the result tables get updated, the changed result is written into an external sink called output. there are three modes of output, append, complete and update. In append mode, only the new rows appended in the Result Table since the last trigger is written to external storage. Append mode is applicable for a scenario where existing rows in the result do not need to be changed. For example, the map operation on any stream. In complete mode, the entire updated Result Table is written to the external storage. The storage connector decides how to handle the writing of the entire table. In update mode, only the rows that were updated in the Result Table since the last trigger will be changed in external storage. (“Structured streaming programming-model” 2019) .

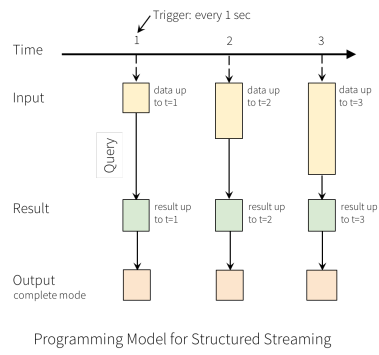

Figure 15. Spark Structured Streaming Programming Model ("Structured streaming programming-model" 2019)

5 Comparative Analysis of streaming Frameworks

In this chapter, the frameworks presented in chapter 4 are compared according to several features that are essential to streaming processors e.g windowing Mechanism, messaging delivering and processing guarantee, state management, fault tolerance mechanism. Similarly, the frameworks are also compared to some other features e.g data sources, supporting programming languages, SQL support, etc.

5.1 Windowing Mechanism

Samza supports time windows, session windows, and global windows ("Samza Windowing" 2019). A global window is a single infinite window over the entire message-stream. Samza supports both fixed and sliding time windows.

Samza only supports processing time windowing but with integrating Apache beam, Samza also supports event-time windowing. ("Samza core-concepts" 2019).

Flink supports time-based windows, session windows, and global windows. Flink supports both fixed and sliding time windows. Flink has support for both processing time and event time windowing. ("Flink Windowing" 2019)

Storm has also support for different windowing. Storm supports both fixed and sliding time windowing. Storm does not have support for session windowing. Storm has support for global windowing. Storm has support for both processing time and event time windowing ("Storm Windowing" 2019).

Structured Streaming has support for fixed and sliding windowing. Spark Structured Streaming does not have support for session windowing and global windowing. Structured Streaming has support for both processing and even time windowing. ("Structured streaming Windowing" 2019)

Windowing Feature	Apache Samza	Apache Flink	Apache Storm	Apache Spark Structured Streaming
Fixed Windows	yes	yes	yes	yes
Sliding Windows	yes	yes	yes	yes
Session Windows	yes	yes	no	no
Global windows	yes	yes	yes	no

5.2 state management

State management is an important feature of streaming frameworks because frameworks require to do stateful computation on data streams. Examples of stateful computations are count and join operations. These kinds of operations require state and are not idempotent operations. Different streaming engines provide different state management mechanisms. Samza implements the state management by a technique called state-store where each Samza task has its own state database. The state-store is associated with a particular Samz's task and stores data corresponding to that task. Also, Samza stores the state locally on the same computer on which the task is running. This is because to achieve the performance by not making remote queries over the network. Also, the state is stored on disk so that more state can be fit in than memory. Any storage engine e.g LevelDB, LMDB can be used for storage but Samza ships with a key-value store engine built on top of RocksDB. Having state storage locally with a Samza job on the same computer and storing it on disk brings a great advantage in Samza, especially for a task where there is a large state storage. ("Samza statemangement" 2019)

Spark Structured Streaming maintains states in both continuous model and micro-batch. The state is maintained using two external storage systems: a write-ahead-log that supports durable, atomic writes and a state-store that can store a large amount of data durably and allowing parallel excess e.g S3, HDFS. (Isah et al. 2019)

Storm has support for both in-memory and disk-based storage for maintaining state. Storm has a default in-memory based state implementation. Also Storm has a Redis backed persis-

tent state implementation. (Isah et al. 2019) (“Storm Statemangement” 2019)

Flink supports both local and external storage for state management. Local storage for state management is either manage in local memory or on disk with a key-value database such as RocksDB. Also, External storage is also used for maintaining state in Flink Examples of external storage are HDFS, S3. (Isah et al. 2019) (“Flink Statemagement” 2019)

Streaming engine	State management
Apache Samza	key-value,local storage, RocksDB
Apache Flink	key-value. local storage, external storage, RocksDB,S3
Apache Storm	key-value, in-memory,local storage,redis
Apache Spark Structured Streaming	externl storage, write ahead, statestore

5.3 processing guarantee

Samza guarantees at-least-once processing (“Samza Processing Gurantee” 2019). Samza ensures that a message will be processed at least once but might be reprocessed more than once e.g reprocessing the streams in-case of failures. Samza ensures that no data is lost but a duplicate might still be possible. Samza achieves at-least-one semantic by a concept called checkpointing. Samza expects the input system to meet the following requirements 1) An incoming stream is divided into one or more partitions. Each partition is independent of each other. Each partition is replicated across multiple partitions to continue the availability of stream in-case of a machine failure.

2) Messages in each partition are in sequence and having a fixed order. Each message has an offset which uniquely identifies the position of the message in the partition. Similarly, messages are consumed sequentially within each partition.

3) A Samza job can start consuming the sequence of messages from any starting offset. Kafka or Kafka like system ensures these requirements.

A Samza task consumes messages from only one partition of a particular stream. The task has the current offset of each input stream (the offset of the next message to be read from stream partition). Every time the task consumes a new message the current offset moves forward. Samza periodically checkpoints the current offset for each task instance. If a Samza container fails, Samza restarts the process on another machine and resume consuming from where the failed process left off. Samza achieves this by looking the offset in the most recent checkpoint and start consuming messages from the offset. In case of an unexpected failure, the most recent checkpoint may be slightly behind the current offsets because before writing the last checkpoint a task may have consumed more messages. In that case, some messages may be reprocessed again. This is how Samza guarantees at-least-once processing.

Spark Structured Streaming ensures exactly-once processing. It ensures exactly-once guarantee through check-pointing and write-ahead log. offsets of the streaming data processed are saved in checkpoints and write-ahead log. In case of failure reprocessing can be started from the beginning but it will not process the acknowledged data. (Isah et al. 2019) ("Structured streaming exactly-once" 2019)

Flink supports exactly-once processing guarantee ("Flink Processing Gurantee" 2019). Flink supports it by using checkpointing and recovery algorithm Flink snapshot the latest state of the processing to durable storage. In-case of failure Apache Flink restores the latest snapshot from durable storage. Flink's snapshot algorithm is based on a technique introduced in 1985 by Chandy and Lamport ("ververica exactly-once" 2019). (Isah et al. 2019)

Storm ensures at-least-once processing guarantee Although Trident (An abstraction on top of Storm) storm can supports exactly-once processing guarantee. Storm relies on acking mechanisim for ensuring at-least-once processing guarantee. (Isah et al. 2019) ("Storm Processing Guarantee" 2019)

Streaming engine	processing guarantee
Apache Samza	Exactly-Once
Apache Flink	Exactly-Once
Apache Storm	At-Least-Once and Exactly-Once through Tradient
Apache Spark Structured Streaming	Exactly-Once

5.4 Fault tolerance

Streaming Engines need to recover quickly from failures to resume processing. This mechanism is called fault tolerance. Samza achieves fault tolerance by saving every change to a separate stream called changelog. In case of a failure, recovery of processing can be made by reading data from the changelog. Usually, a log-compacted Kafka topic is used as a changelog. It retains the most recent value of each key. Also in case of failure Samza tries to reschedule the tasks on the same hosts to reuse the snapshot of local-state saved on the same host. This feature is called host-affinity. With host affinity, Samza does not need to reseed the local state from the changelog. if the state is large reseed can take a long time and for real-time processing, it does not work. ("Samza fault tolerance" 2019)

Flink implements fault tolerance mechanism by a combination of checkpointing and stream replay. A checkpoint is a snapshot of the input stream along with the state for each operator at a specific point. in case of failure, streaming data flow can be reprocessed from the checkpoint by restoring the state of operators and replaying the events from the checkpoint's point. ("Flink fault tolerance" 2019)

Spark Structured Streaming implements fault tolerance mechanisms by checkpointing and write-ahead-log. The write-ahead-log keeps track of which data has been processed and written to the output sink reliably while checkpointing hold snapshot of the state of operators. In case of failure in Structured Streaming, it tracks which state it has last updated in the log and recomputes the state from the data starting from that place. Both for checkpointing and write-ahead-log external storage such as HDFS, S3 can be used. ("Structured Streaming faul

tolerance" 2019) (Isah et al. 2019)

Storm uses acking mechanism for fault tolerance. In case of failure, the acking mechanism replays the storm tuples. The acking mechanisim tracks the completion of each tuple tree with a checksum hash. The value of checksum is zero if all the tuples are successfully acknowledged. In Storm, topology has a "message timeout" associated with it. if every message in a spout fails to process in timeout times, Storm considers that tuple has been failed and replays it. Similarly, the mechanism periodically checkpoints the state of bolt to ensure that it can be restart from the saved state if it needs to restart. (Isah et al. 2019)

Streaming engine	Fault tolerance
Apache Samza	change-log, host-affinity,
Apache Flink	checkpointing, stream replay
Apache Storm	checkpointing, stream replay
Apache Spark Structured Streaming	checkpointing, write-ahead-log

5.5 Other Comparisons

Samza applications can only be written in Java currently. It offers built-in support for different data sources like Apache Kafka, AWS Kinesis, Azure Event Hubs, Elastic Search and Apache Hadoop. Also, it's quite easy to integrate with other sources. Samza can read streams from AWS Kinesis but can't write to it. Samza has some limited support for SQL Also. Samza SQL only supports simple stateless queries including selections and projections. ("Samza Official" 2019)

Flink applications can be written in Java, Scala and Python. It offers built-in support for different data sources like Apache Kafka, AWS Kinesis, Apache Cassandra, RabbitMQ, Apache Nifi, Twitter Streaming API, Google Pub/Sub, Elastic Search and Apache Hadoop. Flink has support for SQL Also. ("Flink Official" 2019)

Spark Structured Streaming applications can be written in Java, Scala, Python and R It offers built-in support for different data sources like Apache Kafka, AWS Kinesis, AWS S3, Azure Event Hubs and Delta Lake. Structured streaming has support for SQL also. ("Structured

streaming Programming-guide" 2019)

Storm applications can be written in Java It offers built-in support for Apache Kafka, HDFS, Hive, Solr, Cassandra, RocketMQ, JMS, JDBC, MQTT, Redis, Events Hubs, Kinesis, Kestrel, MongoDB, OpenTSDB, PMML and Elasticsearch Storm has also support for SQL. ("Storm Official" 2019)

other Features	Apache Samza	Apache Flink	Apache Storm	Apache Spark Structured Streaming
Programming Language	Java	Java, Scala, Python	Java	Java, Scala, Python, R
Data Sources	Kafka, Eventhubs, AWS kinesis etc	Kafka, Eventhubs, AWS kinesis etc	Kafka, RabbitMQ / AMPQ, AWS kinesis etc	Kafka, AWS S3, AWS kinesis, EvenT Hubs etc
SQL Support	Samza SQL	Flink SQL	Storm SQL	Spark SQL
Deployment	Standalone, Hadoop YARN	Standalone,Hadoop YARN, Apache Mesos, AWS,Docker, Kubernetes, MapR,GCE	Apache Zookeeper	Standalone, Hadoop YARN, Apache Mesos, kubernetes

The comparison between streaming engines is made on some of the important features of a streaming engine. From the comparison, we can't say which one is the best streaming engine because every streaming engine has some strengths and some limitations too. So How to choose a streaming framework? The answer is it depends on use-cases. A streaming framework might fit more than the other one for a certain use-case. for example familiarity with a certain stack e.g if a company has experts in R language then Spark Structured streaming might be a good choice because it has support for R language. Similarly, if some once is already using Kafka as a message broker than Samza might be a good choice because Kafka comes out of the box with Samza. Similarly, if someone needs to work with twitter streaming API the storm might be a good option as it has built-in support for Twitter-streaming API.

Another scenario is if someone needs both batch processing and stream processing then Flink or Spark Structured Streaming would be good options because they both have batch processing support also. Similarly for a machine learning Analytic Flink and Spark Structured Streaming might be good options because of their built-in machine learning libraries. similarly, if exactly-once semantics for processing the messages is an absolute necessary then Spark Structured Streaming and Flink would be the options as they both guarantee exactly-once semantics.

6 Conclusion

There are many applications that require real-time processing of large volumes of data. Extracting meaningful timely insight from unbounded data is very challenging. Earlier Batch processing was used to get insights from these unbounded data by collecting it in batches. Batch processing is very useful to process Big data but they are not useful in those use-cases where results are required in real-time. Streaming engines were designed for these kinds of applications where data needs to be processed in real-time. This thesis explained streaming frameworks and introduced some streaming engines. The important features of Streaming engines are explained. The main contribution of this thesis is to make a comparison between Apache Samza, Apache Flink, Apache Storm and Apache Spark Structured Streaming based on the important streaming engines features like windowing, processing guarantees, fault tolerance mechanism, state management and some other features like programming languages support and data interaction tools. Also, another contribution of this thesis was to introduced Apache Kafka and explain the architecture of Apache Kafka.

Bibliography

Alkatheri, Safaa, Samah Abbas, and Muazzam Siddiqui. 2019. "A Comparative Study of Big Data Frameworks". *International Journal of Computer Science and Information Security,* (): 8.

"Apache Kafka". 2016. Accessed May 2016. `http://kafka.apache.org/documentation.html#introduction`.

Carbone, Paris, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. "Apache flink: Stream and batch processing in a single engine". *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36 (4).

Dean, Jeffrey, and Sanjay Ghemawat. 2008. "MapReduce: simplified data processing on large clusters". *Communications of the ACM* 51 (1): 107–113.

Dendane, Youness, Fabio Petrillo, Hamid Mcheick, and Souhail Ben Ali. 2019. "A quality model for evaluating and choosing a stream processing framework architecture". *arXiv preprint arXiv:1901.09062.*

"Flink fault tolerance". 2019. Accessed December 2019. `https://ci.apache.org/projects/flink/flink-docs-stable/internals/stream_checkpointing.html`.

"Flink Official". 2019. Accessed December 2019. `https://flink.apache.org/`.

"Flink Processing Gurantee". 2019. Accessed December 2019. `https://flink.apache.org/features/2018/03/01/end-to-end-exactly-once-apache-flink.html`.

"Flink programming-model". 2019. Accessed December 2019. `https://ci.apache.org/projects/flink/flink-docs-release-1.9/concepts/programming-model.html`.

"Flink Statemagement". 2019. Accessed December 2019. `https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/state/state.html`.

"Flink Windowing". 2019. Accessed December 2019. https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/windows.html.

Garg, Nishant. 2015. *Learning Apache Kafka, Second Edition.* 2nd. Chapter one. Packt Publishing. ISBN: 1784393096, 9781784393090.

"Hadoop Official Website". 2020. Accessed January 2020. https://hadoop.apache.org/.

"HDFS Design". 2020. Accessed January 2020. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

Isah, H., T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan. 2019. "A Survey of Distributed Data Stream Processing Frameworks". *IEEE Access* 7:154300–154316. ISSN: 2169-3536. doi:10.1109/ACCESS.2019.2946884.

Jay, Kreps. 2013. "About Logs and Kafka". Accessed May 2016. https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying.

———. 2016. "Log Compaction". Accessed June 2016. https://cwiki.apache.org/confluence/display/KAFKA/Log+Compaction.

"Kafka in a nutshell". 2020. Accessed January 2020. https://sookocheff.com/post/kafka/kafka-in-a-nutshell/.

"Kafka Introduction". 2016. Accessed June 2016. https://kafka.apache.org/documentation/#compaction.

"Kafka Official". 2016. Accessed June 2016. http://kafka.apache.org/documentation.html#replication.

Kleppmann, Martin. 2016a. *Designing data-intensive applications.* Chapter 5. O'Reilly Media.

———. 2016b. *Making Sense of Stream Processing.* Chapter 2. O'Reilly Media.

"NameNode - HADOOP2". 2020. Accessed January 2020. https://cwiki.apache.org/confluence/display/HADOOP2/NameNode.

Neha, Narkhede. 2013. "Kafka Replication". Accessed June 2016. https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Replication.

"Oracle goldengate". 2020. Accessed January 2020. https://www.oracle.com/middleware/technologies/goldengate.html.

"Oracle Xstream". 2020. Accessed January 2020. https://docs.oracle.com/cd/E11882_01/server.112/e16545/xstrm_intro.htm#XSTRM72647.

"Samza core-concepts". 2019. Accessed December 2019. https://samza.apache.org/learn/documentation/latest/core-concepts/core-concepts.html#time.

"Samza fault tolerance". 2019. Accessed December 2019. https://samza.apache.org/learn/documentation/latest/architecture/architecture-overview.html#fault-tolerance-of-state.

"Samza Official". 2019. Accessed December 2019. http://samza.apache.org/learn/documentation/latest/core-concepts/core-concepts.html.

"Samza Processing Gurantee". 2019. Accessed December 2019. https://samza.apache.org/learn/documentation/latest/core-concepts/core-concepts.html#processing-guarantee.

"Samza statemangement". 2019. Accessed December 2019. https://samza.apache.org/learn/documentation/latest/container/state-management.html.

"Samza Windowing". 2019. Accessed December 2019. https://samza.apache.org/learn/documentation/latest/api/high-level-api.html#window-types.

"Storm Official". 2019. Accessed December 2019. https://storm.apache.org/releases/2.1.0/Tutorial.html.

"Storm Processing Guarantee". 2019. Accessed December 2019. https://storm.apache.org/about/guarantees-data-processing.html.

"Storm Statemangement". 2019. Accessed December 2019. https://storm.apache.org/releases/2.1.0/State-checkpointing.html.

"Storm Windowing". 2019. Accessed December 2019. https://storm.apache.org/releases/2.0.0/Windowing.html.

"Structured streaming exactly-once". 2019. Accessed December 2019. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#fault-tolerance-semantics.

"Structured Streaming faul tolerance". 2019. Accessed December 2019. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#fault-tolerance-semantics.

"Structured streaming Programming-guide". 2019. Accessed December 2019. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.

"Structured streaming programming-model". 2019. Accessed December 2019. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#programming-model.

"Structured streaming Windowing". 2019. Accessed December 2019. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#window-operations-on-event-time.

Toshniwal, Ankit, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. "Storm@twitter". In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data,* 147–156. ACM.

"ververica exactly-once". 2019. Accessed December 2019. https://www.ververica.com/blog/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink.

Wang, Guozhang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. 2015. “Building a replicated logging system with Apache Kafka”. *Proceedings of the VLDB Endowment* 8 (12): 1654–1655.

Wingerath, Wolfram, Felix Gessert, Steffen Friedrich, and Norbert Ritter. 2016. “Real-time stream processing for Big Data”. *it-Information Technology* 58 (4): 186–194.