UNIVERSITY OF TARTU

Institute of Computer Science

Computer Science Curriculum

Mansur Alizada

Real time vs micro-batching in streaming data processing: performance and guidelines

Master's Thesis (30 ECTS)

Supervisor: Pelle Jakovits,Ph.D

Tartu 2021

Real time vs micro-batching in streaming data processing: performance and guidelines

Abstract:

Data is used in every second of our life. Nowadays, the majority of this data is coming through the Internet. For providing better fast and scalable service, technologies needed to be efficient and scaled regarding those needs. The initiative of this thesis to provide simple workload for engine comparison. In this master thesis, I will focus on Apache Flink, Spark Streaming, Apache Kafka, Apache Storm, Storm Trident for real-time and micro-batch in streaming data processing. This thesis aims to show the comparisons among those technologies.

Keywords:

Stream processing, Apache Kafka, Apache Spark, Apache Flink, Real-time streaming, Micro-batch processing.

CERCS:

P170 Computer Science, Numerical Analysis, Systems, Control

Reaalaeg versus micro-batching voogedastusandmete töötlemises: jõudlus ja põhisuunad

Lühikokkuvõte:

Andmeid kasutatakse meie eludes igal ajahetkel. Tänapäeval liigub suurem osa andmetest läbi interneti. Selleks, et võimaldada kiiremaid ning vastupidavamaid teenuseid, tuleb tehnoloogilised valikud teha neile nõuetele kohaselt. Lõputöö eesmärk on võrrelda valitud andmeedastuse raamistikke. Käesolevas magistritöös võrreldakse Apache Flink'i, Spark Streamingut ning Apache Kafka't vahetuks mikroteenustel põhinevaks andmetöötluseks. Sellega soovitakse näidata eelnimetatud raamistike erinevusi ning võrdlust Lambda ning Kappa arhitektuuride vahel.

Võtmesõnad:

Andmevoo töötlemine, Partii töölemine , Jõudlustest, Apache Flink, Apache Spark, Apache Storm, Apache Kafka, Kafka Stream, Lambda ning Kappa arhitektuuride.

CERCS:

P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

1 Introduction

Nowadays, data is growing exponentially, around us, which is coming from different sources. As more data is flowing into our system, providing better service for customers, we have to process, analyze, make decisions based on the data, because data is valuable at the time it arrives. In this scenario real-time streaming comes into the scene. In real-time streaming data is processed as it arrives. For example, fraud detection, stock prices(trading) are based on real-time data streaming. Another approach for data processing is batch processing. In batch processing, data is collected in a period of time, hours, days, then gathered data is processed. For example, it is used for sending bills to customers monthly. Fortunately, there are many real time and batch processing frameworks that are open source and able to process data. Let's look through some differences between real-time and batch processing for better understanding.

In real-time processing, processors should be available for the response in a short time, in batch processing, the processor gets busy within defined time intervals. Due to that, complete time of the task is crucial in real-time, whether in batch it is not necessary. There is no time limit in batch processing, however, in real-time within specified time limit operations should be finished. In batch, data is collected in a period of time and going to be processed in batches, in real-time random amount of data is processed in time. Last and one of the important difference between them, batch processing is less expensive and simplest processing way for business applications, whether real-time processing requires complex architectural design and high quality hardware specifications.

Different scenarios demand various data collecting and processing structures. In this thesis, I will mainly focus on real-time and micro-batch, compare them in different frameworks, compare different frameworks' performance for real-time and micro-batch, based on finding the most famous stock index case and guide for better results regarding to these frameworks. In this master thesis, I will focus on Apache Kafka and discuss Apache Flink, Spark Streaming Apache Storm, Storm Trident and Apache for real-time and micro-batch in streaming data processing.

1.1 Motivation

In streaming data processing, the crucial point is how to handle this data without losing any valuable part of it, in a short time frame. As Tyler Akidau [1] explains Big Data as it is something, which needs faster processing capacity than relational database systems and to achieve useful output from it, you have to have better structure for processing it. Based on our purpose, data processing provides for us functionalities to manipulate data and get useful information for our business related tasks for customer services within high performance and low cost.

1.2 Research problem

Due to increasing interest in real-time and micro-batch data streaming, many different kind of stream processing engines are developed by companies. However, exact guidelines for choosing the right framework for the specific business cases are changing from performance

perspective for one framework to another. The main purpose of this thesis, compare real-time and micro-batch streams in different kinds of frameworks, analyze outputs from the experiment, look through performance improvement methods and dive into architectures that are using real-time and micro-batch streaming.

1.3 Structure

This thesis consists of eight chapters. In the first chapter, I share some general information about the topic, main concept of the key points and about research problem. In the second section of this thesis, background information is provided about streaming and architectures which are used by different companies. Third section of the thesis talks about the frameworks, necessary information which every reader should know at least in beginner level and usage of these frameworks at different companies. Next section is about related experiments, which has been done on some of the frameworks. That section provides results of each experiments. For fifth section, I researched about advantages and disadvantages of frameworks and other functionalities which are necessary to cover. Sixth section is about experiments and their results and guidelines for frameworks.

In the end, I made conclusion for the thesis and future works that can be done.

2.Background

Before getting started with the research, I would like to let readers know about the technologies that I used in this thesis. Firstly, I will discuss about some terminologies which is in the thesis's topic and data-processing architectures, then I will talk about the technologies which are used for this thesis.

2.1 Streaming

Nowadays, everything we see on the internet or in the technology world is data, and that data is being transmitted from one point to another one without cease. Transactions we do through our bank's mobile app, 'likes' and 'comments' of our social media accounts, messages and millions of other things are examples of the data being generated as a stream. These data are called as big data, which is growing rapidly, unbounded, infinite and impossible to handle with traditional database systems. Big data is described as 4V [4]. Volume, velocity, variety, and veracity. **Volume**: is the size aspect of big data. Big amount of data, from business perspective is good to forecast much more exact predictions than less amount of data. **Velocity**: is the speed of the data that is flowing to the system. As data increasing in volumes, application should be able to process the data quickly as close as to real-time. This refers to velocity of big data. **Variety**: data is coming from different kind of sources; data format could be different from in each source, which can be classified as structured and unstructured data types. Structured data formats have orders, where unstructured data formats don't own them. This variety of unstructured data makes issues for storage, mining and analyzing data. **Veracity**: refers quality of data. It means, how useful and reliable is this data for the business, where meaningful records are kept.

As I already mentioned in the velocity aspect of Big Data, data needs to be processed quickly in real-time or near real-time. This process is called data streaming. Data streaming is a speed-focused approach, which is processing data continuously. Data continuously generated by different sources is a stream data. There are two types of streaming; Bounded and Unbounded [5]. A bounded stream is a stream, which has end and unchanged data, where everything is known about data. An unbounded stream is a stream, which never ends (or we do not know when it will), and event data is being processed continuously, which means future events are not important for the events that are being handled now. In this case, all events are similar, and the only way to differentiate them is their creation or received time. In contrast, start and end times are defined in bounded streams. Data is handled in batches, and thus, these kind of streams are also called batch processing. While streaming experiments, latency and throughput are two main factors to evaluate performance.

Real-time: Real-time streaming systems take inputs continuously, process them and give an output of the input. These systems differ from others with time attributes, which try to respond within a very short time or in a specific time limit. Therefore, it takes care of delay in a response, which helps to detect threats or patterns in real time for different kinds of scenarios for business. E-commerce, online bookings, credit card fraud prevention, GPS, radar like systems are based on real time data processing. As disadvantages of real-time data processing systems, we can say

that, these systems' design are very complex and expensive processes. Below I would like to share following crucial qualities which are mentioned by Facebook. [2]

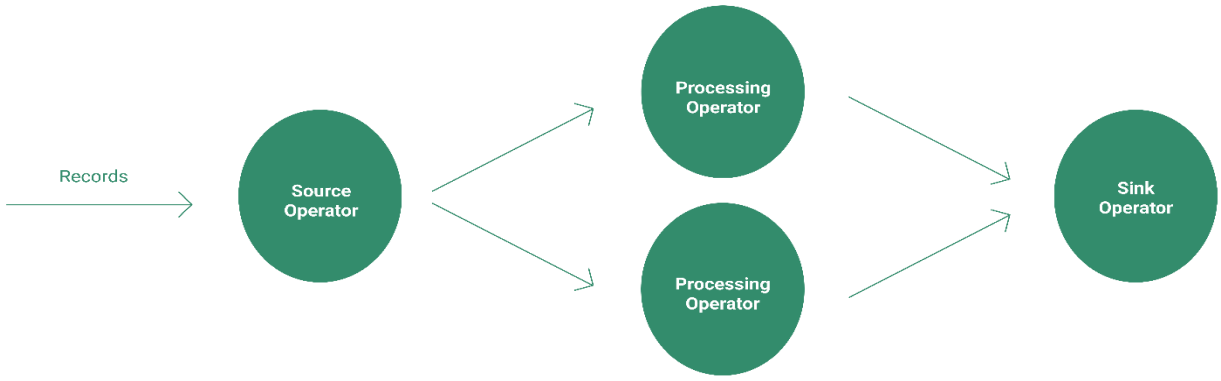


Figure 1. Real-time stream processing model.

- **Ease of use:** Usage of technologies, programming languages and deployment should be considered.
- **Performance:** Latency and throughput are 2 important factors in processing to be taken into account.
- **Fault-tolerance:** We should be aware of kinds of failures and recover the system.
- **Scalability:** Changes should be done without much more efforts and easily.
- **Correctness:** Should all of the data entered be considered for output?

Micro-batch: It is mainly used for getting high throughput. In micro-batch processing, small batches of data are accumulated then, these groups of data are processed within small time intervals like 500ms or 5 sec [3]. This process is executed repeatedly in some time intervals. While one process is executed, another stream of data is collected. Downside of the micro-batch processing is higher latency, which occurs while waiting time due to accumulating micro-batches.

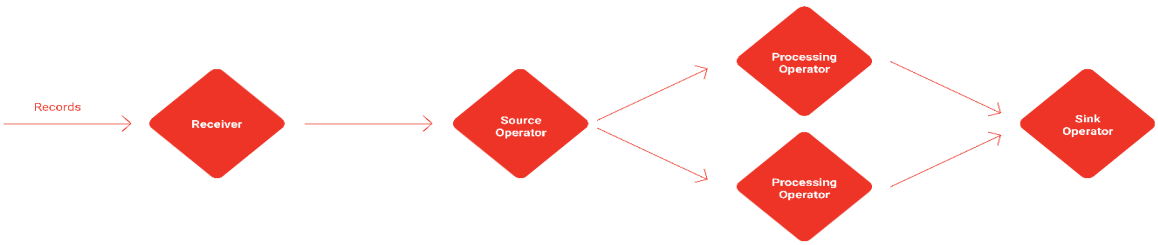


Figure 2. Micro-batch stream processing model.

2.2 Use cases of streaming

Streaming data processing is one of the hot topics today. Let's look through for 2 reasons in which we need streaming.

1.Business wants to have timely data with low latency.

Trading system, where user buys/sells equities or FX orders on the market, is one of the case that requires latency under 20ms.

2.Infinite volumes of data handling.

User tracking on the websites, website performance analyses are the example of infinite volume of data handling.

Event-driven applications: event is given a definition that an event is an occurrence within a particular system or domain [6]. It is something that has happened, or is contemplated as having happened in that

domain. The word event is also used to mean a programming entity that represents such an occurrence in a computing system. An application, which is fed by sensors with data and this application detects and reports an event for further analyzing and reactions is one of the example for event-driven applications. For example, Uber, Bolt are example of event-driven application, where user searches for a taxi and events occur behind of the schema and responses are suggested back to the user.

Streaming analytics: extracting useful information from raw data, it could be a historical data from databases too, analyze, predict or perform other necessary operation on the data and report. In industry, streaming analytics process helps organizations, to understand and track their clients better, promote them different campaigns based on their stored data and develop products for customer's need. Streaming analytics applications are commonly used for monitoring quality of cellphone networks, user behavior in mobile applications.

2.3 Data streaming architectures

In this thesis, I will look into the 2 most famous architectures in data streaming, for better guidelines in real-time and micro-batch stream data processing.

Lambda Architecture represented by the Greek letter λ. Main aim of Lambda Architecture was having robust, fault-tolerance and low-latency based architecture. This architecture takes advantage of both batch and stream processing's methods. Lambda Architecture serves to solve computation of arbitrary functions [7]. It has 3 layers. Batch view = function (all data). Real-time view = function (real-time view, fresh data). Query = function (batch view, real-time view). Figure 9 illustrates Lambda architecture.

Batch layer: it stores copy of the master dataset and precompute batch view on that dataset, corruption of that dataset is irreparable. That is why choosing the correct storage for batch layer is important. This dataset is updated constantly. The result of the batch layer in the form of batch view is transferring to the serving layer. As machine learning algorithms take too much time for training the model and give a better result, this computation takes place at batch layer. As time consuming task happens in batch layer, this layer has a higher latency. From performance perspective of batch layer, amount of resource when new data is updating batch view and produced batch view size is important.

Serving layer: it loads the batch views in somewhere (in database) so that they can be queried. Serving layer updates itself automatically when new batch views are available and they are tightly connected. From performance perspective, latency and throughput are important factors here.

Speed layer: purpose of this layer process last few hours' data, which is not presented in the batch view. It is similar to batch layer, however it only looks through recent data, whereas batch layer looks through all and compensate high latency of batch layer. At the cost of higher latency, with micro-batching stream processing, you can achieve fault-tolerant accuracy in speed layer, latency will be hundreds of milliseconds to seconds. In micro-batch stream processing, small batches of data are processed at a time, if anything fails, the whole batch will be replayed and batches' order is strict.

Usage scope of Lambda architecture.

Log analytics and log ingestion are one of the common use cases for this architecture. Internet of Things is another use case, which is used by enterprises, where a huge amount of data is coming from sensors and is handled by this architecture. Real-time sport analysis is also a usage area for Lambda Architecture, where the batch layer prepares various operations according to the game and present to the viewer. Recommendation engines, marketing campaigns are also built in this architecture. Twitter, Stack Overflow, Flickr, Walmart have implemented this architecture indoors [49].

Advantages of Lambda architecture.

- Data is immutable in Lambda Architecture, which means it stores in a raw format. After processing or transformation original data is still accessible. It also protects data from corruption, hardware failure or other issues which can cause data loss. Implementing new business cases in this case is not hard.

- As I mentioned in the previous chapter about recomputation, it is another advantage of Lambda, which helps to make fault tolerance without much more effort. For example, due to bug or data loss, it is always possible to recompute, reprocess data.

- Because of the layers, using different kinds of technology stack and distributing responsibilities into well-defined blocks which are not tightly coupled with each other

make it another advantage of this architecture. Technology stack can be changed with others without any troubles.

- Connecting a new source for data feeding is easy in Lambda Architecture.

Disadvantages of Lambda architecture.

- As having advantages because of 3 layers, developing on these layers is considered complex. Synchronization between 2 layers needed to be managed in low cost and effort.
- Setting up this architecture in production environment required quite good hardware components.
- Doing the same job in batch and speed layer is being considered as duplication.

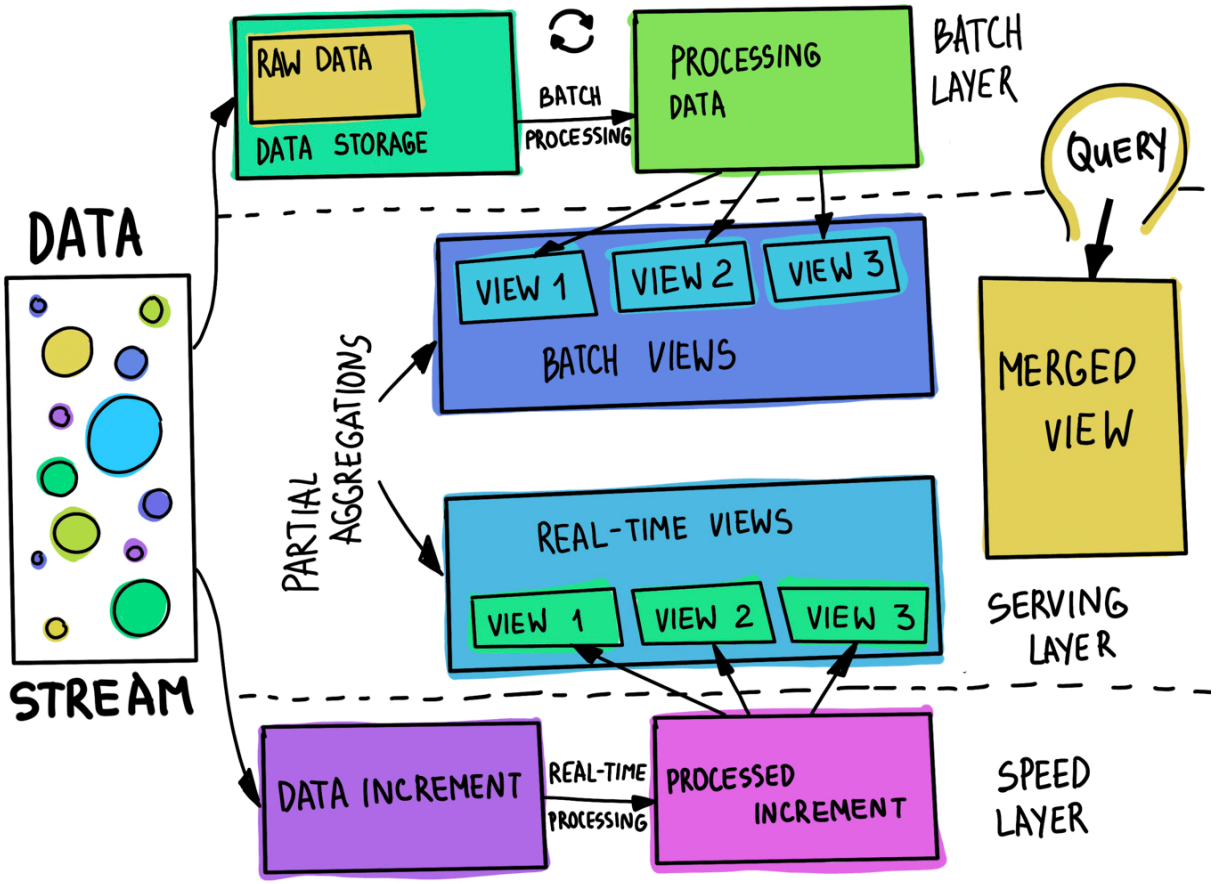


Figure 3 – Lambda architecture [9].

What is Kappa Architecture?

Few years later after Lambda, Jay Kreps presented another approach for real-time stream processing [10]. Main idea behind Kappa Architecture is that it is possible to do real-time and batch stream processing with one stream processing framework. In Kappa architecture, incoming data is going to fed into stream processing engine or real-time layer, which is responsible for running jobs and real-time data processing. Later the data fed into to serving layer where it queries any results.

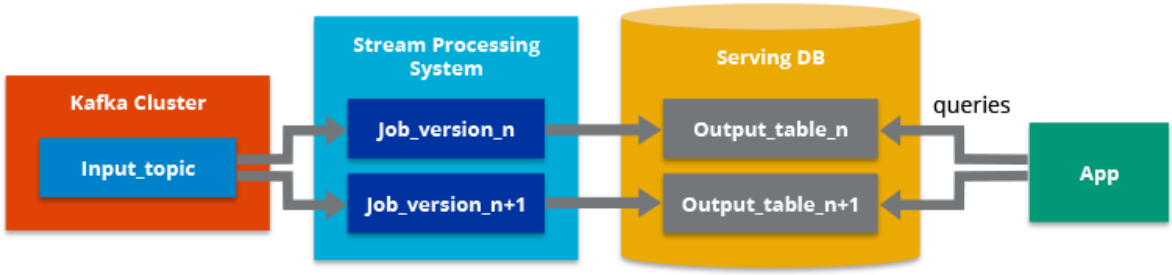


Figure 4 – Kappa architecture [11].

Usage scope of Kappa architecture.

Kappa architecture is mainly used for real-time monitoring, reporting and dashboarding. As a possibility of using Flink, Spark makes implementing ML in this architecture easier and use for fraud detection systems. For example, Uber is using Kappa for this purpose [50]. Flink is based on Kappa architecture.

Advantages of Kappa architecture.

- Only one code base to maintain with a unique framework.
- Reprocessing is needed when code changes.
- It is not complex compared to Lambda, which is the main weakness of Lambda and main reason for creation of Kappa.
- Information is always up to date.
- Because of reprocessing during code changes, identifying business changes is easy.

Disadvantages of Kappa architecture.

- As core of Kappa Architecture is message broker such as Kafka, it is not a really data warehouse, problems such as single node partition size, retention and cost compared to HDFS are problems in this case.

- Low volume topics get full faster [50]

In conclusion of this chapter, decision of application's data pipeline architecture, framework it depends on characteristics of the application that you are going to build. For example, when you are going to apply the same algorithm for the real-time streaming and old historic data it would be nice to apply Kappa Architecture. If you are expecting different outputs from real-time and historical data, Lambda Architecture can solve this problem. Table 1, illustrates comparison of Lambda and Kappa architectures. Apart from these architectural patterns, other architectural alternatives also existed such as Zeta, Iot-a. For choosing the best architectural pattern for real-time streaming, you should make a clear decision which answers questions such as what kind of data structure you have? What kind of processing do you need? What are the business objectives? How much can you afford for hardware resources?

	Lambda architecture	**Kappa architecture**
Nature of data	Immutable	Immutable
Data processing capability	Real-time and batch	Real-time
Delivery guarantees	At least once	Exactly once
Reprocessing time	In every batch cycle	Only when code changes
Layers	Batch, real-time, serving	Streaming and serving
Scalability	Scalable	Scalable
Fault-tolerant	Fault-tolerant	Fault-tolerant
Storage	Permanent	Temporary

Table 1. Comparison of Lambda and Kappa architectures

3. Stream processing frameworks

In this chapter, I will discuss the stream processing frameworks, their architectures and some of the APIs of these frameworks.

3.1 Apache Kafka

Apache Kafka is one of the most famous distributed streaming platforms developed in 2011 by LinkedIn. Mainly, is used for handling huge amounts of data in real time. As documentation [12] says, it has 3 main capabilities: publish and subscribe to stream, store streams, process streams. From application perspective, it is used as building real time data pipelines which retrieve data between applications, or react to the data streams in real time. As other stream engines, it can run on clusters too. Before going into architecture of Kafka, I would like to talk about another specific features of Kafka. Kafka is a distributed system, which means that, it stores, processes, sends data to different nodes, which is called broker.

3.1.1 Architecture of Kafka

Before going into Kafka, let's review some of the terms that are important.

Topic - a stream of messages or records belonging to a category. Data is stored in the topics and consumer consumes data from the topic. Topic can have many consumers from it [13]. For making the data fault tolerant in Kafka, every topic can be replicated across regions, data-centers, where multiple brokers keep copy of data. The unit of replication is topic partition.

Broker - manages storing the data in the topic or topics. Kafka Server is a synonym for broker. Brokers are responsible for receiving messages from the producer and storing them. More than one broker is acting as Kafka Cluster. In this case, one broker will control broker failures. Kafka brokers are provided with retention for the topics, such as life expectancy of the topic is 7 days, or certain size is 1GB.

Producer - sends a message to the topic or in another way, it publishes messages to the topic, it is like a data source. Kafka provides 3 methods of sending messages to the topic.

- Fire and Forget: There is no acknowledgment whether sending is successful or not.
- Synchronous send: waiting for the message' success.
- Asynchronous send: using callback function sending the message, and gets an error from Kafka broker for handling.

Let's look through configuration parameters for configuring producers in Kafka from high importance to low.

Serializers- defined format of the data we are going to write to Kafka, **acks-**default value is 1, with the help of this parameter, producers can control the durability of the messages written to Kafka, for optimizing the durability acks is setting as acks=all. It can sometimes lead to higher

latency. Another option for having a durability is configuring **retry** and **delivery. time.outs** which are used for retrying the request in case of failures. **retry**.backoff.ms. it sets the duration between two **retry. buffer memory size** - important for keeping messages for delivery, we can define timeout(max.block.ms) for blocking buffer's availability., **compression type**- default is none, by defining it, we can reduce network utilization and storage.

Consumer - subscribe to the topic and read data from the topic or topics. Below listed configuration parameters which are important for optimizing the consumer's performance in cases such as event failure, for tuning to handle highest throughput level and etc.

Bootstrap.servers is a comma separated list of host and port pair for connecting to a Kafka broker. You can configure multiple bootstrap.servers in the configuration, if one fails, then it falls back to another one. Deserialization is used on consumer side of Kafka, for converting bytes of array into desired data type. We can configure consumers to fetch less times by increasing **fetch.min.bytes,** for example, default value is 1,it waits as soon as 1 byte of data is generated, we can increase this amount ,however it will cost us additional latency.,**heartbeat.time.interval -** frequency of the time interval for sending to broker, **session.timeout.ms** - amount of the broker needs to get at least one heartbeat from the consumer. Heartbeat should be ⅓ of session timeout, which is used for detecting the consumer failures. If no heartbeats are received before the expiration of it, then broker removes the consumer from the group. With the help of an **offset**, every message and track of each messages that consumer has received so far can be accessible. First message has an offset zero, then it increments by one. If there is problem with the server, consumer can read from where it was before.

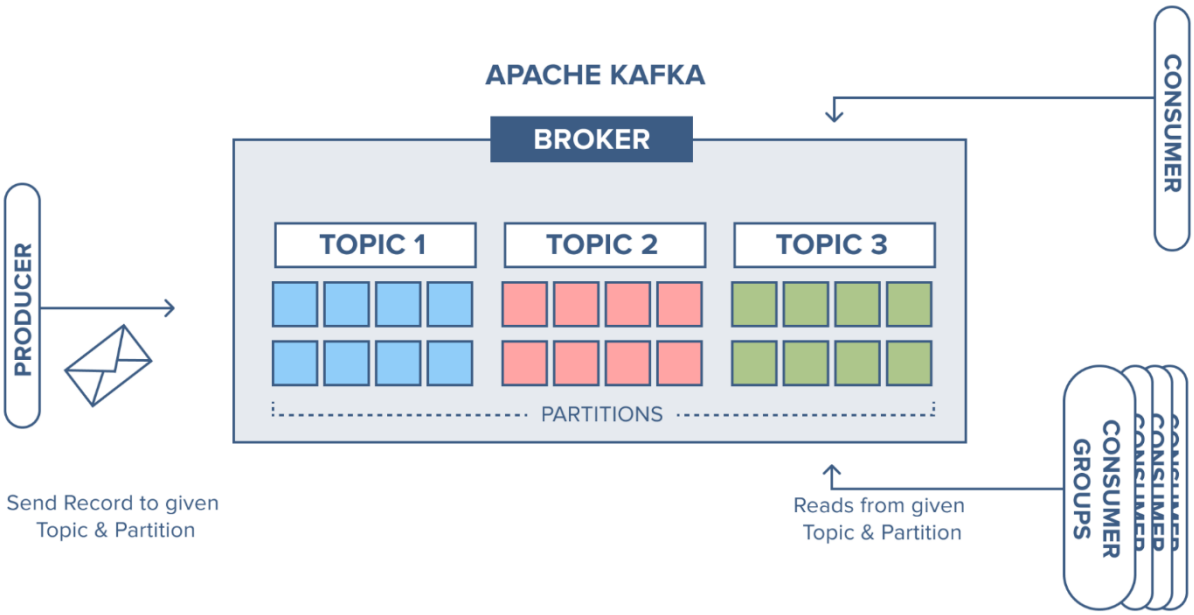


Figure-5. Record Flow in Apache Kafka [14]

Kafka has 5 core apis [15].

Producer API: allows applications to send data to the topic in the Kafka cluster. It is Kafka Producer class, as previously I mentioned, some configuration parameters can be used within help of this class.

Consumer API: allows applications to read data from the topic, it is Kafka Consumer class.

Streams API: allows applications to receive data from input streams and transfers to output topics. Core api for streaming in the Kafka. One of the main objects of this API is KStream.

Connect API: for connecting data source to Kafka and outputs to other application systems.

Admin API: for controlling, topics, brokers and other Kafka objects.

As I mentioned before, the producer sends data to the topic. While sending data, it chooses the partition on based ProducerRecord key [16]. It is a key/value pair to be sent to Kafka. This consists of a topic name to which the record is being sent, an optional partition number, and an optional key and value. A key is used for partitioning data. Also, the key can include some meta data on the actual value of the message that helps you control the subsequent processing. Data with the same key goes to the same partition. Keys can be a string value, Avro messages or depending on your configuration. Your partition strategy depends on your data and kind of processing you are trying to do. Kafka achieves fault tolerance by replicating each partition over number of servers. By default, Kafka uses Round-Robin strategy. Kafka Producer has three mandatory properties which needs to be configured: **bootstrap.servers**: list of host, which producers uses for connecting to Kafka cluster, **key.serializer**: name of class, which is used for serialize the keys of records, **value.serializer:** name of the class, which is used for serialize the value of records. Consumers, subscribe to topics, read records from partitions and write result to another system. Kafka Consumers are part of Kafka Consumer Group, which means, many consumers are subscribed to a topic. Each of these consumers will receive from various subset of partitions in the topic. Common technique for scaling data in Kafka is adding more consumers to consumer group for high-latency. Single consumer, can't provide better performance in some cases, that is why adding more consumers one of the method of scaling, which provides load balancing. In conclusion, you create consumer group for each services that you want to read messages from topics. What happens when consumer crushes or shuts down? In this case, partitions will be used by other consumers on the group. Changing partition ownership is called rebalance. One of the important topic in Kafka is rebalancing of consumers, which should be taken care of. For creating consumer in Kafka, KafkaConsumer instance is using generally. It is almost similar as KafkaProducer. Bootstrap.servers, key.deserializer and value.deserializer are main properties.

Kafka has many configuration parameters [17]. Number of configuration parameters can be listed as: 208 Broker configuration parameters, 27 Topic configuration parameters, 65 Producer, and 70 Consumer configuration parameters. Before going to production it is always recommended to test within different configuration parameters for the best result, however it is time consuming and highly an expensive experiment.

Before going into to Kafka Streams [18], I would like to summarize things about Kafka.

- High-Throughput-Kafka is able to handle high-velocity and high-volume data, with proper hardware.
- Low Latency-Kafka response within very low latency of milliseconds.
- Fault-Tolerant-If any machine failures happen, it doesn't affect the workflow.
- Durability-Data is never lost within Kafka it stores messages in the disk.
- Distributed-multiple producers and consumers can publish and retrieve messages at the same time.

3.1.2 Kafka Streams

Kafka provides stream processing from version 0.10.0 [18]. Apart from the things, I mentioned above, stream-processing library is one of the important parts of Kafka. It helps developers to develop their application, producing, consuming, processing events without using external processing frameworks like Spark, link and others. It is possible to use it either via low-level Processor API or via the Kafka Streams DSL.

Kafka Streams is a client library. One of the reasons to compare other streaming frameworks. It provides millisecond latency, distributed joins, aggregations, DSL, distributed processing and fault-tolerance with fast failover, reprocessing capability and etc. Kafka Streams is focused on Balance the processing load (as we mentioned in Architecture of Kafka), maintain local state for tables, and recover from failures. For example, if application fails it can always look up its last partition from Kafka and process from the last offset it committed before failing. Even if, local state is lost, it can always recreate it from logs. It is similar how consumers are handling failures.

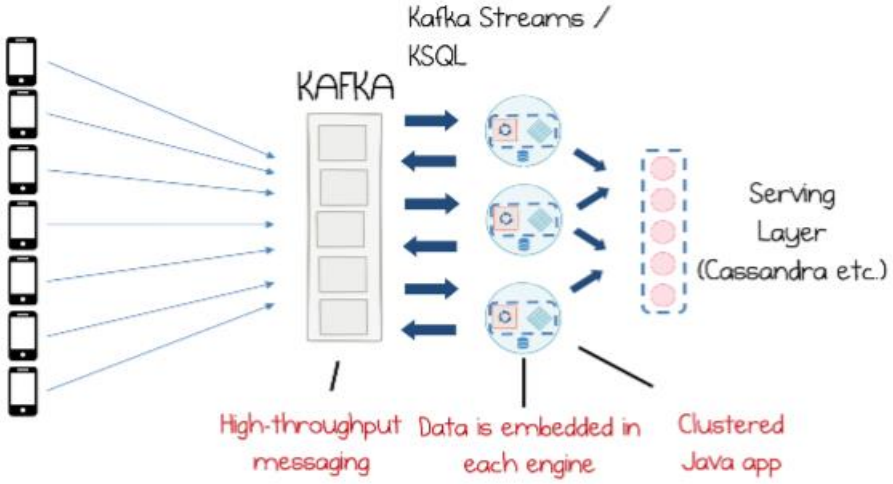


Figure – 6. Kafka Streams implemented application [19]

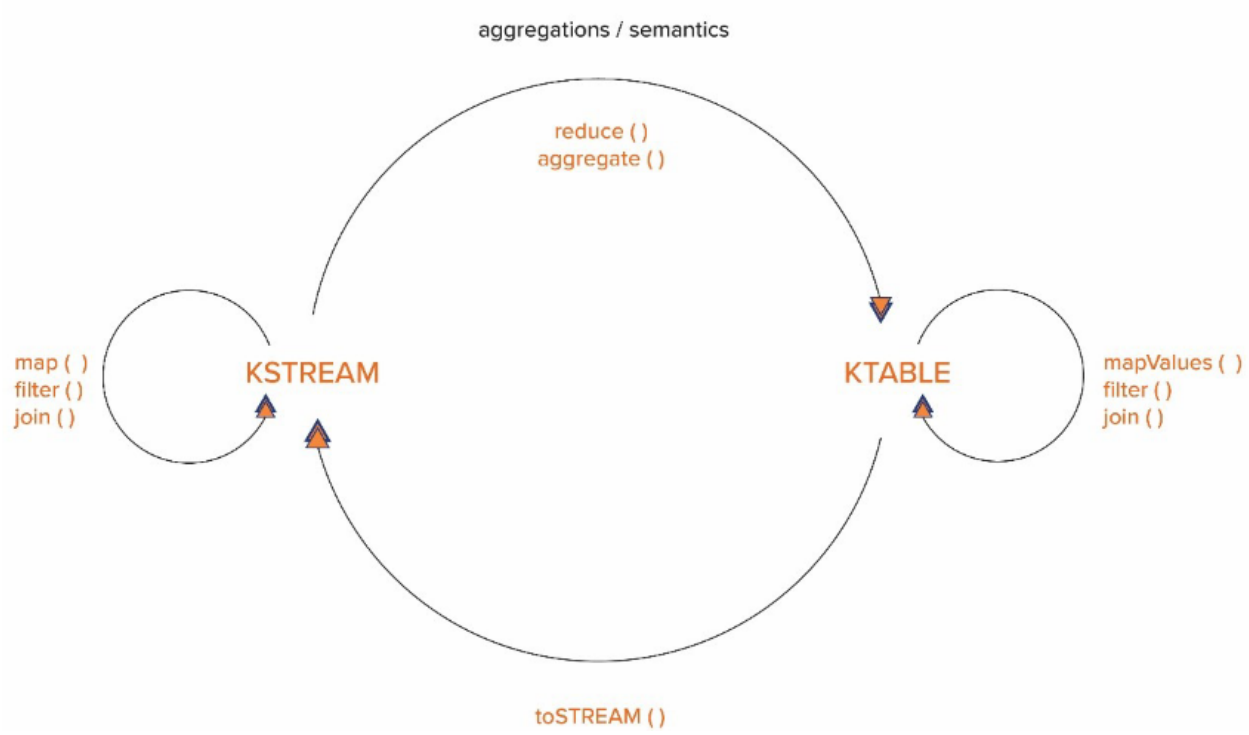


Figure –7. Kafka Streams DSL.

Kafka Streams DSL is built on top of Streams Processor API. It is built in for streams and tables in the form of KStream, KTable. KStream is an abstraction of a record stream, where each data is a key/value pair. It provides functionalities such as map, filter, join and etc. Ktable is an abstraction of a changelog, where record is an Insert or Update(Upsert) depending to the key if any rows exist with the same key. Figure 7 illustrates Kafka Streams DSL.

3.2 Apache Flink

Apache Flink is an open source framework and distributed processing engine for data streaming by The Apache Software Foundation [20]. It is the next generation Big Data tool also known as 4th generation of Big Data [21]. Flink has been designed to run in all common cluster environments, perform computations at in-memory speed and at any scale [20]. It is an alternative of MapReduce, it processes data more than 100 times faster than MapReduce. It can use HDFS to read, write, store the data. Like other engines, it also runs in clusters. Flink supports Java, Scala, Python either using DataStream or DataSet APIs.

For stateful stream processing, Flink supports multiple notions of time. **Event time**: is the time, when an event is generated at the source. Events from the sources store time in their metadata. It is possible to extract this time from the event. Flink uses this time, when there is a need for a computation based on that time. **Processing time**: is system time of the machine that is executing the operation. An hour of processing time window includes all records that get into the operator between the times that included an hour time frame. It is simple notion of time and provides best performance and latency. **Ingestion time**: is the time that event enters to Flink.

Conceptually, it is in-between event time and processing time. Default one in Flink is processing time. However, event time provides more accurate results, as it includes timestamp at the time when an events is produced by the source.

3.2.1 Architecture of Flink

Flink is used in Batch, Interactive, Real-time, Graph, Iterative, In-memory processing. It works on Kappa architecture. Flink setup consists of many processes, which run across in multiple machines. It is integrated with cluster resources managers, like Apache Mesos, YARN and Kubernetes, however, it can be also configured as standalone cluster. Main components of Flink are JobManager, a ResourceManager, a TaskManager and a Dispatcher. All of these components run on JVM. Let's look into what these components do.

JobManager: it controls execution of the application. Each of the applications is controlled by different job managers. JobManager accepts application which consists of JobGraph, and jar file which has libraries and other sources. In conclusion, it leads distributed execution.

ResourceManager: responsible for managing TaskManager slots. While JobManager requests TaskManager slots, ResourceManager provides a TaskManager with slots to provide the JobManager.

TaskManager: is the worker process of the Flink. Each of them provides a certain number of slots. JobManager can assign tasks to those slots to execute them. TaskManager exchanges data with other TaskManagers.

Dispatcher: while job executions, it provides a REST interface to submit applications for executing.

Apart from those, Flink provides a mechanism, which recovers the state of applications. This mechanism, helps when there is failure on the application. For this, Flink uses checkpointing algorithm. The algorithm doesn’t pause the application, but decouples checkpointing from processing. Other mechanism is savepoint, with that, you can pause an application and resume it later or go back in time. With the help of checkpointing Flink achieves fault tolerance.

3.2.2 Flink’s APIS

Depending on the data source, if you are going to do batch or stream processing, Flink provides 2 kinds of APIS. DataSet API is used for batch, where DataStream API is used for streaming.

DataSet API: within help of this api we can filter, map, join, group data sets reading from data sources, write via sinks to desired place [22]. Map, FlatMap, Filter, Join, Rebalance, Union, Distinct are examples of transformation functions.

DataStream API: is used in real time data streaming for filtering, updating, windowing, aggregating, etc. [23]. In the beginning, data streams are coming from message queues, socket streams, files. Results are retrieving via sinks to data files or to third party applications.

Table API and SQL: is used for unified stream and batch processing. With help of this api user can write complex SQL queries in an easy way. Using table environments, with dataset or datastream api tables can be created. After creating table user can select from table in simple way [24].

FlinkCEP: is a library which is used for Complex Event Processing. With this, you can detect event patterns from a stream of events [25].

State Processor API: provides functionality for modifying savepoints and checkpoints using DataSet API. Main concept is improving application, which was previously blocked by parameter or other things, without losing the state of application after restart.

Gelly: It is a graph api, used for graph analysis, for example, create, transform, modify graphs, etc. It provides a map transformation for vertex or edge values.

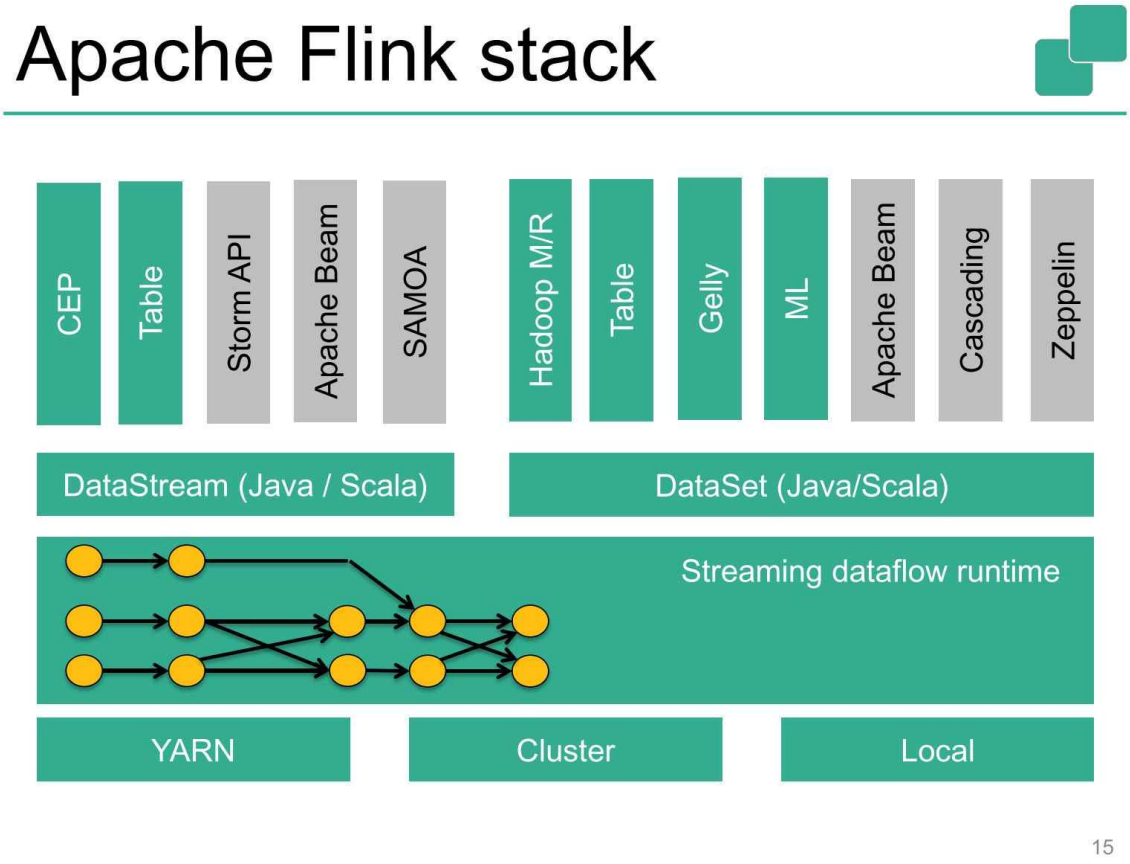


Figure – 8. Apache Flink Stack [26]

3.3 Apache Spark

Apache Spark is a high-performance, a unified, distributed computing engine, open source project by Apache. [27] It supports Java, Scala, Python and R. It is developed on Scala, therefore it is suggested, if you want to get better performance, developing on Scala has better opportunities. Compare to other frameworks, developing on Spark is easier. Apache Spark provides libraries for SQL, streaming, machine learning. As other streaming engines, Spark provides data loading from storage, compute it, however, it doesn't keep data in itself long. In Big Data, Spark is 3G, where Flink is 4G as we mentioned before. It is designed for batch processing. This picture describes overview of Spark.

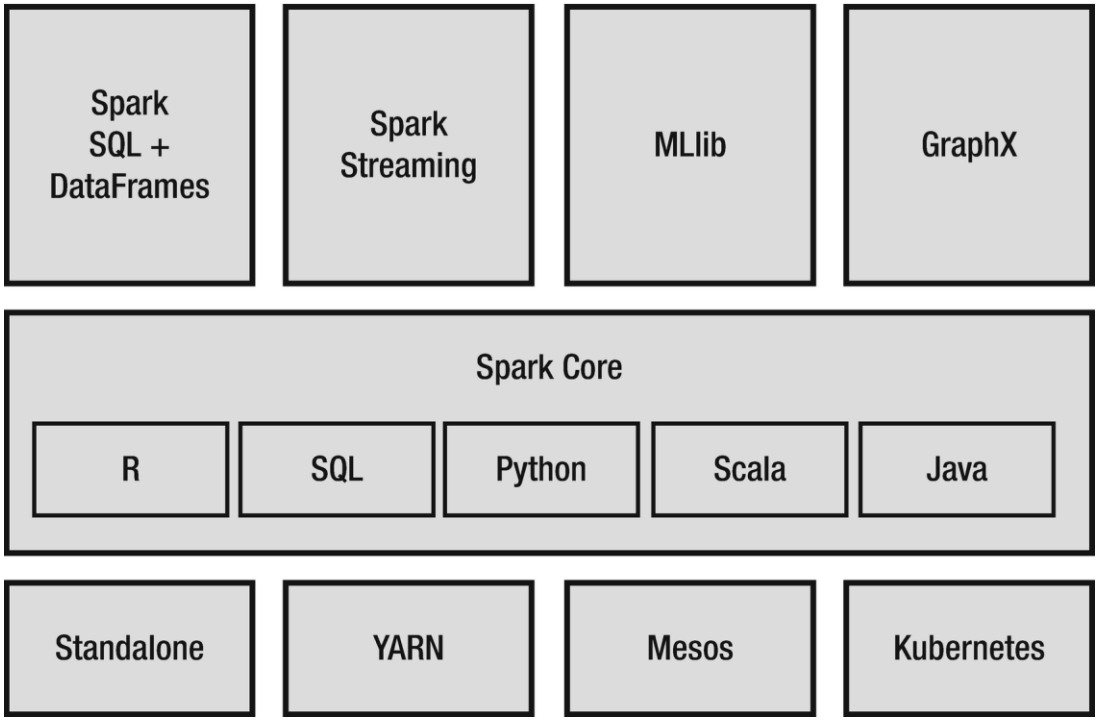


Figure 9. – Overview of Spark [28]

3.3.1 Architecture of Spark

Spark is suggested as an alternative to Hadoop, because, it is faster than Hadoop. It performs computations on JVM. It supports, standalone cluster manager, YARN, Mesos, Kubernetes. Application starts with SparkContext. After starting of SparkContext, a driver and executors start on the worker nodes of the cluster. Generally, Spark applications have two processes: Driver process and Executor process. Driver process is responsible for running application's main method, transforming user input to the tasks, split the tasks and schedules them to run on the executor. It is one of the important part of Spark application and keeps important information regarding application status while application is up. Executor process is responsible for executing

the task which is assigned to it by driver process and send the result of execution to the driver. It provides in-memory storage for RDDS.RDDs are data abstractions and present distributed collections. They are fault tolerant and Spark handles RDDs in parallel. RDD persistence is one of the important factor in Spark, which is caching data in memory while processing. During processing, nodes keep partitions in memory. If anything is lost, it recomputes from the beginning which was created. RDDs are capable of join, reduce, map, read and write data between systems. It is available for Scala, Java, Python. RDDs can be stored in various storage level. [29] Default one is MEMORY_ONLY.In this level, RDDs are storing as deserialized Java objects in JVM. If volume of RDDs can't fit to memory, some of partitions will be lost and they are going to be recomputed on the fly each time. If RDDS partitions are large to be stored in RAM, then it can be written to disk. Of course, it decreases computation's speed, however, it provides better fault tolerance. RDDs are lazy. What does it mean? Spark doesn't start computing, until an action is called. An action is a method that returns data. The picture below describes the action. Action triggers the scheduler, which builds DAG.DAG is directed acyclic graph, which contains the flow operations between RDDs.

3.3.2 Spark APIs

Spark has Structured APIs and lower-level APIs. Each of them has their own interfaces. Structured APIs are: Datasets, DataFrames, Spark SQL. Lower-level APIs are: RDDs and Distributed variables. [30]

Structured APIs are used for batch and streaming. As we mentioned before it has 3 interfaces. Datasets and DataFrames are distributed collections.

DataFrames: we can create data frames from SQL, from RDDs, or from other data sources. It is based on data which has column name and other types of information. It is the same as table in relational database. It has both transformations and actions.

Datasets: are a mixed type of DataFrames and RDDs. You can convert it to RDDs or DataFrames. It is compiler-time type safety. Because of known data structures, datasets provide less memory consumption. Example picture in here.

3.3.3 Spark Streaming

Spark Streaming is part of Spark, which provides functionality for streaming live data with high-throughput and low latency. As other streaming engines, Spark can load data from various data sources like Kafka, Flume, etc. Spark handles data after arriving parallel in the clusters. Data is dividing into micro batches. Data is processing after it converted into RDD sets. This abstraction is called Discretized or D Stream. DStream's data can be published to databases or file systems. It is also allowed to use DataFrames and SQL operations for streaming. As we mentioned before, it is fault tolerant, that means, streaming should continue all the time without any failure, during any failures, the system should recover itself. For this purpose, Spark provides data checkpointing.

Metadata checkpointing-in this case, Spark saves some important information about streams to HDFS for recovering. This information contains: **configuration**: that is used for the streaming, DStream **operations**: operations like map, join, etc. used for streaming, **incomplete batches**: batches which haven't finished its job.

Data check pointing-like metadata checkpoint, it's purpose is also the same, however, in data checkpoint, while streaming, there are possible cases that RDDs are connected with each other, like one depends on previous one, to decrease the risk, it saves RDDs to HDFS in scheduled period of time.

Another thing which, worth to mention is tuning in Spark. In the documentation, [31] they provide tuning techniques for better performance.

3.3.4 Spark Machine Learning Library

Spark provides ML library for machine learning developers. It contains common machine learning algorithms, classifications, regressions, clustering and others. It decreases machine learning's complexities such as infrastructure, configurations etc. For example, using org.apache.spark.mllib.classification you can use SVM,NaiveBayes classification methods within spark.

3.4 Apache Storm

Apache Storm is a free, open source, distributed streaming engine for real time data processing. Same as other distributed engines, it is commonly used in real time data processing,machine learning, distributed computing and etc. It is fault tolerant and easy to configure and run [32].

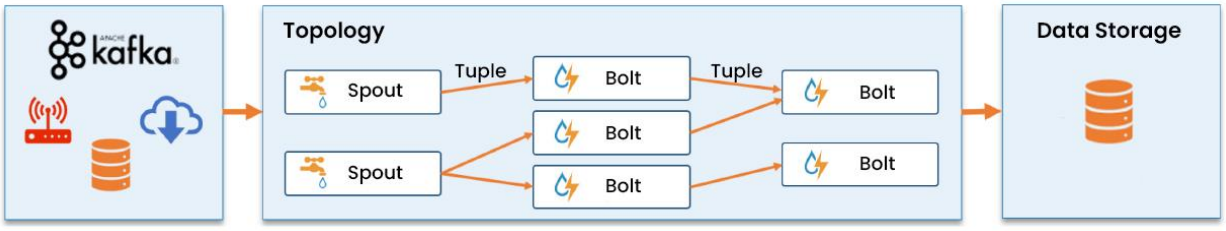


Figure –10. Streaming with Storm [32]

3.4.1 Architecture of Storm

Storm is written in Java and Clojure.For better performance, it is suggested to use Java. In Storm, the work is shared among different components, and all of them are responsible for some specific operations. **Spout** is a component which handles input streams. You can create a spout, to read data from storage systems, or from Twitter API or from queue systems. There are 2 types of spout, reliable-which gets tuple again, if there is a failure, unreliable-which doesn't care about lose. Spout passes the data to the component which is called a **bolt**. Bolt saves data in storage systems, or transfers it to another bolt. Bolts are capable of doing computations or other map, join like functions. Bolts are accepting tuples, and producing tuples as output. Cooperation between spout and bolt is called **topology**. It is similar to MapReduce, however, in Storm, topologies run forever, until the time we kill them. One of the important topic, we would like to highlight for performance perspective is stream grouping. In which, you have to consider about how data is going to be carried between spouts and bolts.

Shuffle Grouping: it is the most used one. It sends tuples to randomly chosen bolts, which takes into consideration that each of them will have the same amount of tuples.

Fields Grouping: it gives an opportunity to check how many fields of tuple are going to be sent to bolt.

All Grouping: it sends a copy of each tuple to all instances of the receiving bolt.

Custom Grouping: you can create your own, custom grouping with the help of CustomStreamGrouping.

As I said before, Storm is fault-tolerant, data won't be lost. There are 2 ways to run Storm. In Local Mode, it runs on JVM. In Remote Mode, we submit the topology to the cluster, which runs in there.

For getting low-latency, high throughput without taking into consideration hardware requirements, it is important to get familiar for some of the settings that Storm provides. **Buffer Size**: there are 2 buffer size, one for spout and bolt executor, another one is for outbound message queue. Very small and very large size is not desirable in both cases. It could make high memory consumption or low memory issues. **Batch Size**: producer batch size, transfer batch size. The batch size for writing into queue any spout/bolt and messages that are going to a spout/bolt correspondingly. For low latency, it is recommended to set it to 1. For high throughput it is recommended to put it to greater than 1.**Wait Strategy**: is used for decreasing CPU usage. It is applied in the following scenarios:

Spout Wait: when we have low data traffic, that there isn't any tuple to emit, we use this technique, which prevents calling nextTuple each time, preserving CPU.

Bolt Wait: again, in the same scenario goes for the bolt wait, when the queue is empty, no need to check an empty queue.

Backpressure Wait: when a spout/bolt is trying to transfer data to the next pair, and the pair's queue is full, it will try to rewrite, this strategy provides some waiting time in between.

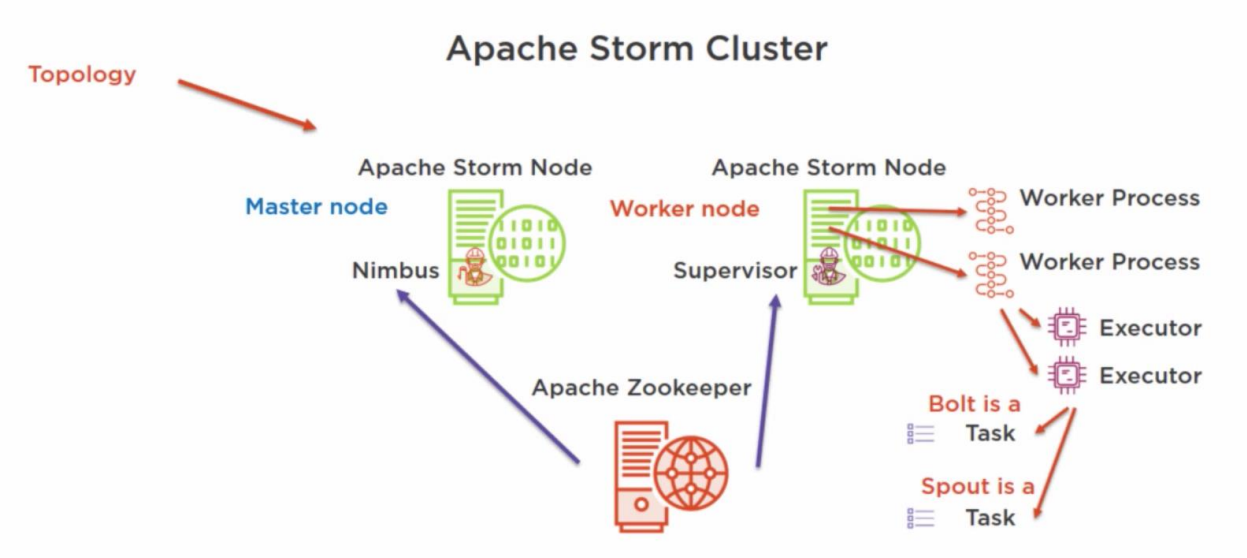


Figure – 11. Overview of Storm [33]

3.5 Usage of frameworks at companies.

In this section, I will discuss usage of frameworks at companies, how they handle their data, amount of data that is processed by frameworks in production environments.

3.5.1 Kafka at companies.

Kafka at LinkedIn.

Kafka is the core part of LinkedIn, as it is developed in-house [43]. It is used as activity tracking, message exchanges, metric gatherings and more. They maintain over 100 Kafka clusters,4000+ brokers, which has 100000 topics and 7 million partitions. Number of daily messages is more than 7 trillion. During 2010, it has 1400 brokers and 1.4 trillion messages per day.

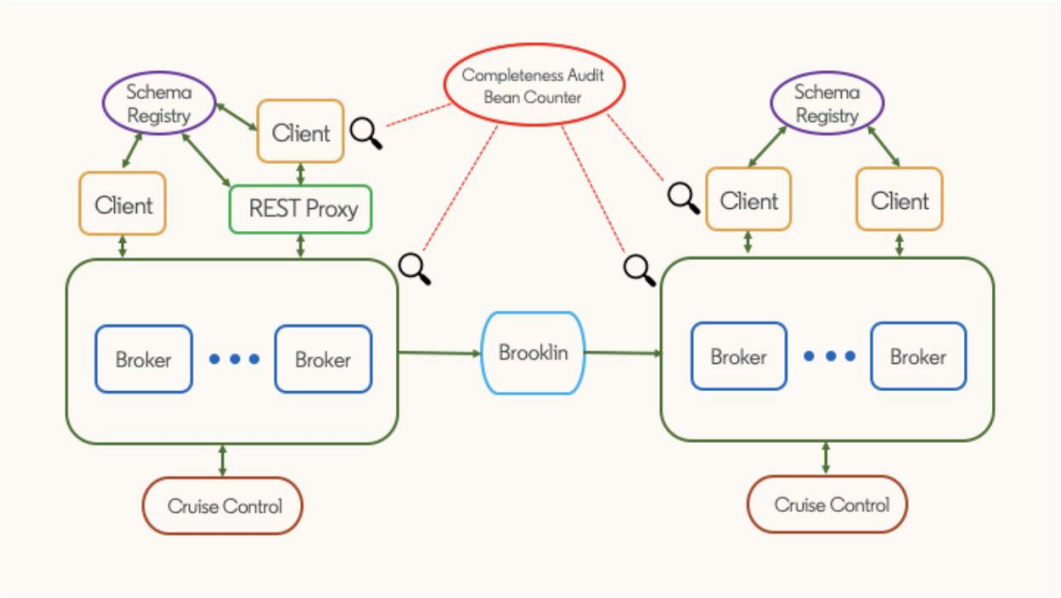


Figure 12. Kafka Ecosystem at LinkedIn [43]

Cruise Control monitors clusters and automatically make adjustments for the resources for meeting the performance goals. With a REST API Cruise Control make interactions with the user. REST API is used for adding brokers add_broker (broker_info), removing broker decommission_broker (broker info), rebalancing the cluster rebalance_cluster (balance_percentage), and etc.

Workload Monitor collects Kafka metrics from the cluster, then from these metrics, it generates a model, which is included CPU utilization, disk usage, network inbound, network outbound. Each workload meant for only one partition, then monitor's output is analyzer's input.

Analyzer is the main part of the Cruise Control, it uses a heuristic analyzer, which can take from few seconds to weeks to propose the optimistic proposals. For having optimal proposal, they made hard and soft goals for achieving.

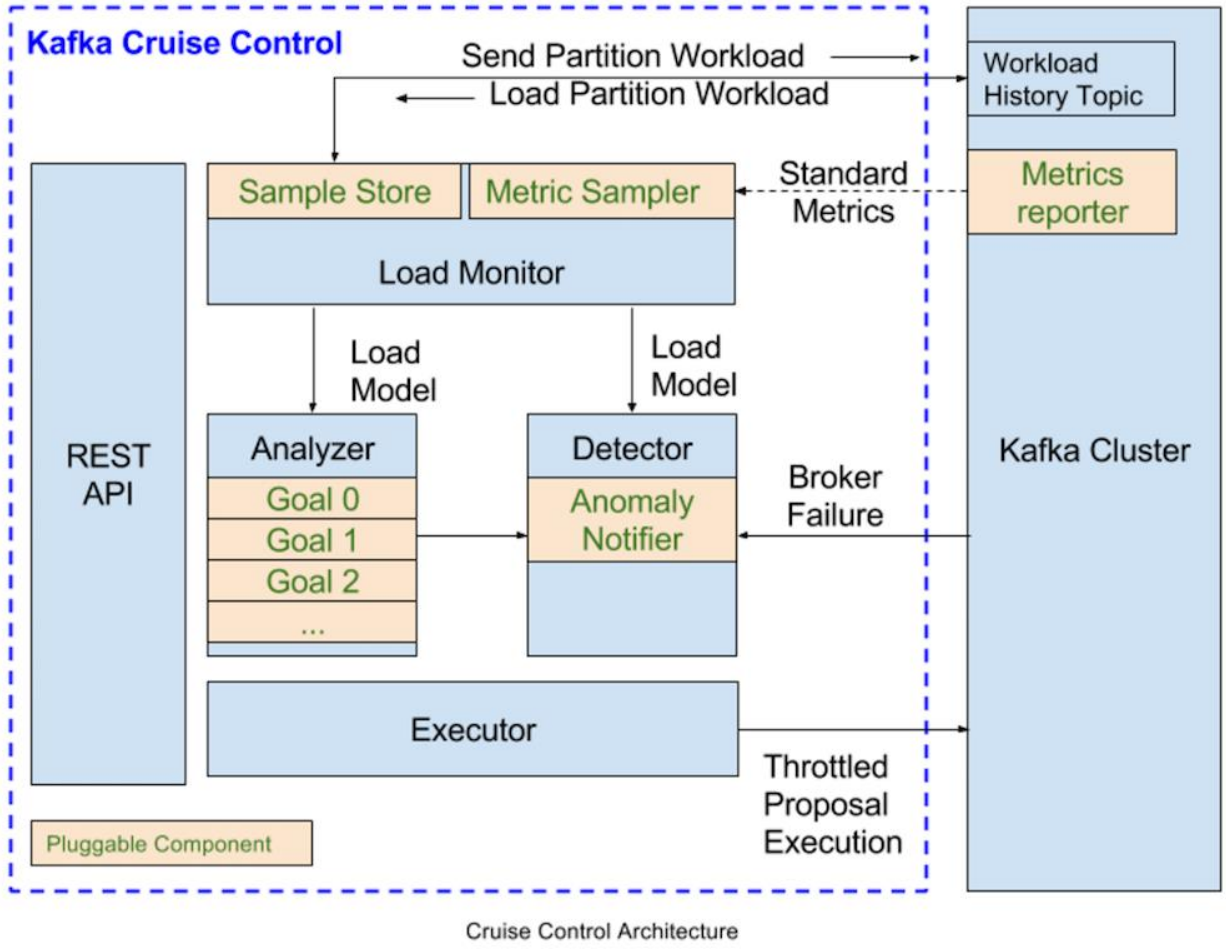


Figure 13. Cruise Control Architecture [43]

Kafka at Uber.

As one of the biggest taxi company, Uber handles a lot of real-time processing in their production environment. They process few petabytes per day within more than 10K topics in a day. Like finding the driver near to customer, estimated time of arrival, auditing is solved by Kafka Streams. They use at-least once delivery guarantee for minimalizing data loss and batching capabilities for high throughput [44].

3.5.1 Flink at companies

Flink at Alibaba.

Alibaba is one of the largest e-commerce platforms in the world using Flink as their search and recommendation engine for real-time streaming. Main reason for choosing the Flink was low latency and high throughput required for the engine. More than trillion events are processed

daily. During peak business hours Flink is capable to handle 472 million transactions per second. Alibaba made their own contributions on Flink and developed incremental checkpoint mechanism, Streaming SQL and etc. [45].

Flink at Zalando.

Zalando is one of the largest e-commerce companies in the Europe uses Flink for real-time process monitoring and ETL. They use Flink to detect special patterns of input streams. As I mentioned in previous chapter, Flink is a good choice for Complex Event Processing, which is the business case in Zalando [46].

3.5.2 Spark at companies

Spark at Databricks.

In 2014, Databricks, sorted 100TB of data on disk in 23 minutes using 207 machines on EC2 which was 3X faster and 10X fewer resources than Hadoop which costed them $451. In 2016, NADSort is presented by Nanjing University, Alibaba Group Inc. and Databricks, which is able to do the same sort in 2983.33 seconds for $144.22 and 3057.67 seconds for $147.82. Purpose of these benchmark was to find best cost efficient way. NADSort consists of stage, the map stage, the shuffle stage and the reduce stage [47].

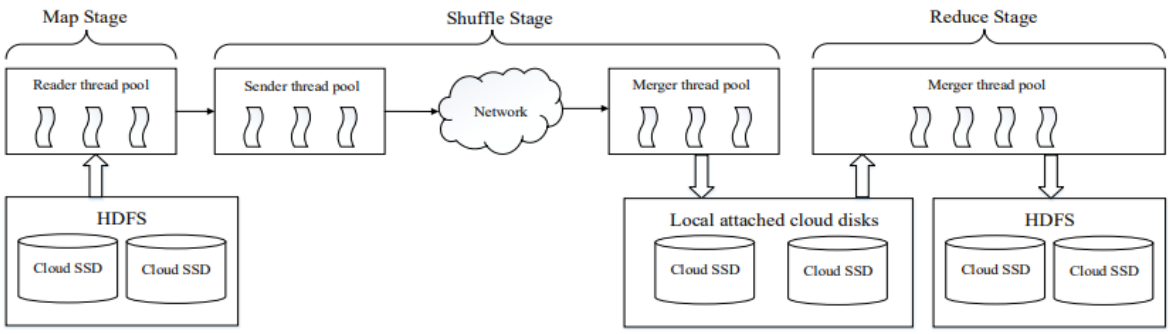


Figure 14. Pipeline of NADSort [47]

Spark at Facebook.

Facebook was using previously Hive for batch analysis. As main reason, Hive was quite resource intensive, Facebook migrated to Spark. In the figure 15, performance comparison of them is shown. 60 TB of compressed data is used and Spark performed 90TB shuffle and sort data within

250.000 tasks in a single job. During this experiment, number of issues found out and solved. For performance bottleneck, Facebook used Spark UI metrics and Jstack [48].

Spark at Walmart.

Walmart uses Spark for data analysis, that, it gets data from HDFS or Hive, generate report within seconds with help of Spark SQL. It also helps to load data from Teradata tables and perform aggregations, calculations on them and publish to Kafka.

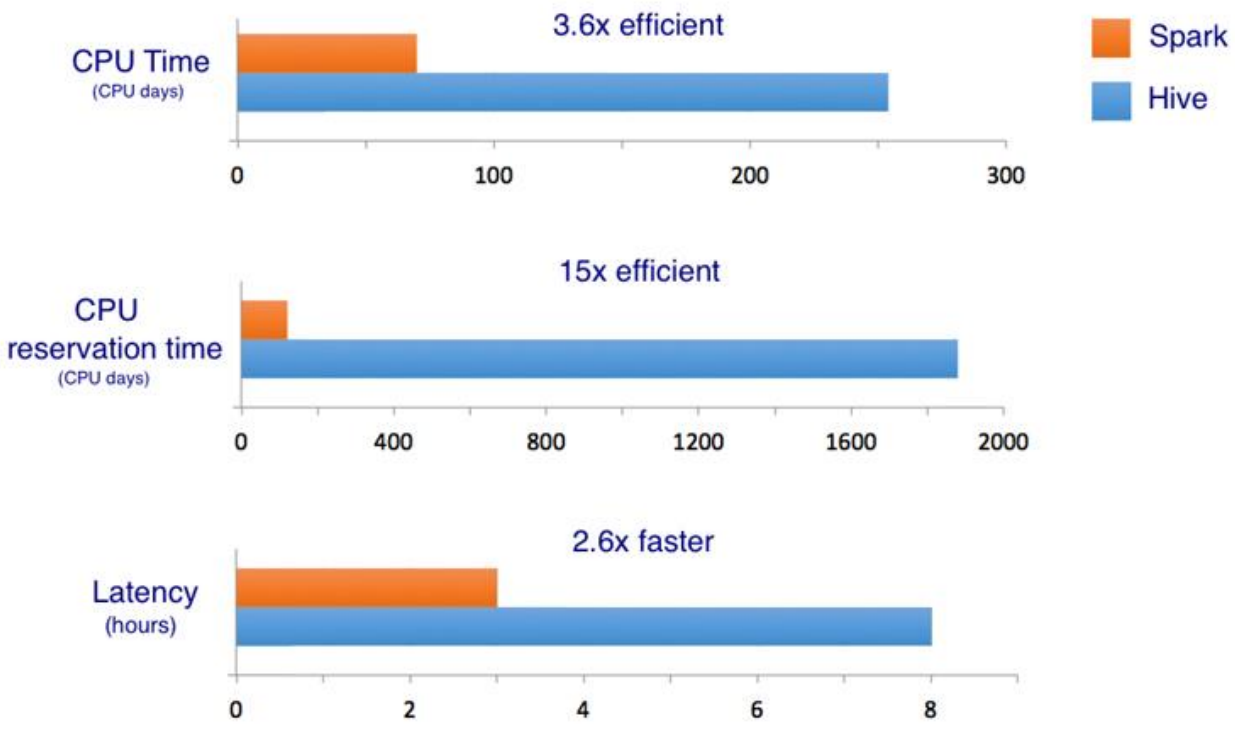


Figure 15. Spark vs Hive performance [48]

4 Related Works.

Before going into the practical part, I would like to give a short overview about what has been done in batch and stream processing. While working with batch or stream processing, companies do beforehand analysis regarding to business plan. Main questions around the problems are performance issues, rich in functionalities, risk factors, cost of the systems, technology compatibility and etc. Due to these questions, we investigated some papers and frameworks, to analyze output such as latency and throughput of the experiments which has been done for testing purposes. Most of the companies like Google, Facebook, Yahoo regarding to their business needs, develop own streaming benchmarks. Before getting into the researches, I would like to share The 8 requirements of real-time stream processing, [34] which is crucial in building streaming application. In the related works and experiment part of this thesis, some of these rules are applied such as 1st, 2nd.

1st rule: Keep the Data Moving

Aim on this requirement is, to achieve low latency as much as possible decreasing useless storage operations, while streams entering to the system. It increases latency. Another problem, we have to avoid spaghetti conditions for processing. Application shouldn't check additional conditionals before processing. From the 1st rule's conclusion, we can say that, in real-time stream processing, data should be processed in fly, without storing them and checking for conditions, which increase workload of CPU.

2nd rule: Query using StreamSQL

For analyzing the data, the stream engine should have its own querying techniques. Traditional SQL isn't sufficient in this case, because streaming doesn't have an end table. Due to streams, which are infinite sequences of tuples that are not all available at the same time and operations over streams must be monotonic, which is hard to solve with traditional SQL. For example, windowing or stream-specific operators.

3rd rule: Handle of Stream Imperfections

In relational databases, like Oracle, MySQL presence of data must be for processing or querying. However, during streaming data appears on fly, we have to have mechanisms for data loss, delayed data handling or missing data handling. Having waiting time infinitely, is the main problem in this requirement. In real-time streaming, time out is a must, because of application should go on with other processing. Time windowing is the solution for those kind of problems, which most of the streaming engines provide. For example, Google Cloud Dataflow solves it with help of watermarks and triggers.

4th rule: Generate Predictable Outcomes

Results of processing should be deterministic and repeatable. From fault tolerant and recovery side, it is important that, replaying and reproducing of the stream should give the same output. For example, Flink provides exactly-once semantics.

5th rule: Integrate Stored and Streaming Data

Regarding business concepts, some of the streaming would require historical data modifications or adding partitions of data to real data for calculating or presenting some historical procedures. From this point of view, integration of stored data with streaming data is crucial. Using client-server connections will add additional latency, it is suggested to use embedded database system.

6th rule: Guarantee Data Safety and Availability

One of the important stages in streaming is failure recovery. This rule is for making sure that application has an active state, there isn't any data loss despite the failures. For example, Flink has Checkpoint mechanism, Kafka has replication factor for this purpose.

7th rule: Partition and Scale Applications Automatically

Within the 7th rule, stream processing should be distributed among multiple machines for fair load balancing. Due to this, it should be possible to split applications over multiple servers for low latency and this process should be transparent.

8th rule: Process and Respond Instantaneously

For achieving high performance, applications should have highly-optimized execution paths, minimal-overhead execution engines.

4.1 Yahoo Streaming Benchmarks

Yahoo Streaming Benchmarks - is one of the open source real-world streaming benchmarks, which was released by Yahoo Inc. in 2016 [35]. Main idea behind this benchmark was to find the best platform for data streaming. In this benchmark, Flink, Storm and Spark are used.

Design of the benchmark: As I mentioned on the above, this benchmark is a real-world advertisement application. For consuming and storing the data Kafka and Redis is used accordingly. Data, which is used in streaming, consists of advertising campaigns and advertisements regarding them. Pipeline of the benchmark is to consume data from Kafka, remove unnecessary fields and save a windowed count of matching events for each campaign into Redis. Work flow is as follow:

1.Consume an event from Kafka

2.Deserialize the JSON string.

3.Remove unnecessary events

4.Take a projection of the relevant fields

5.Merging events, by coherent ids for storing in Redis.

6.Take a windowed count of events for each campaign and store it in Redis with its campaign id and timestamp.

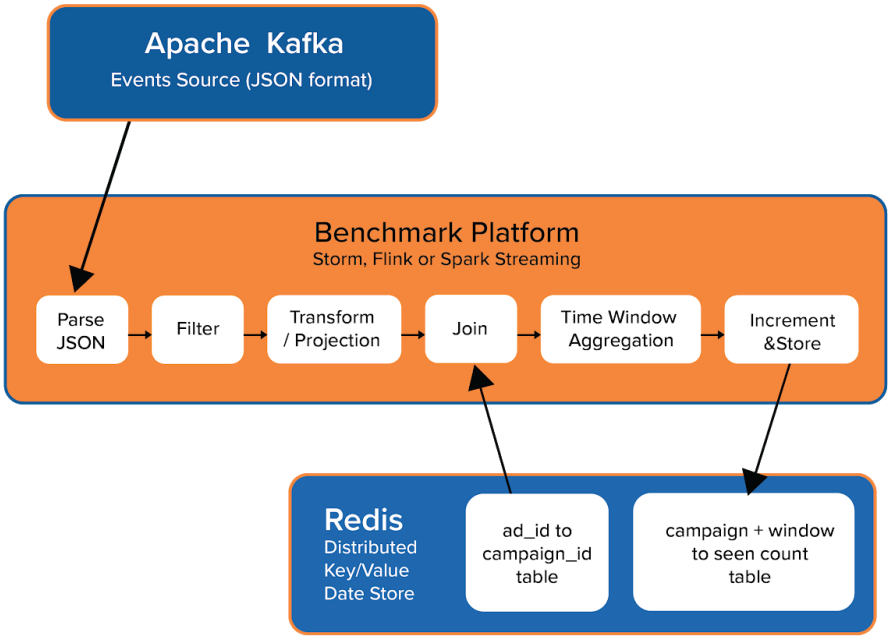


Figure 16. Streaming Benchmark Design [35]

In the setup of benchmark,100 campaigns,10 ads per campaign,5 Kafka nodes with 5 partitions,1 Redis node,10 worker nodes ,5-10 Kafka producer nodes and 3 Zookeeper nodes are used. Each node provided with two Intel E5530 processors running at 2.4GHz and total of 16 cores.17000 events per second created by Kafka producers within 10 instances, with between 25 to 30 nodes. In the beginning of the benchmark's Kafka is cleaned and Redis is deployed with the data. Then, stream engines and Kafka producers started, after 30 minutes, producers were stopped. After

few seconds, streaming job itself was stopped. This benchmark is implemented for each streaming computation engines in three different versions.

Apache Flink Benchmark - was implemented by using Flink's DataStream API. Notable configurations were taskmanager.heap.mb (JVM heap size)[citation] as 15360 and taskmanager.numberOfTaskSlots (The number of parallel operator or user function instances that a single TaskManage).Using FlinkKafka-Consumer to read data from Kafka. Kafka consumes 50k events/sec to 170k events/sec. As we can see from the graph after 99th percentile, latency grows exponentially.

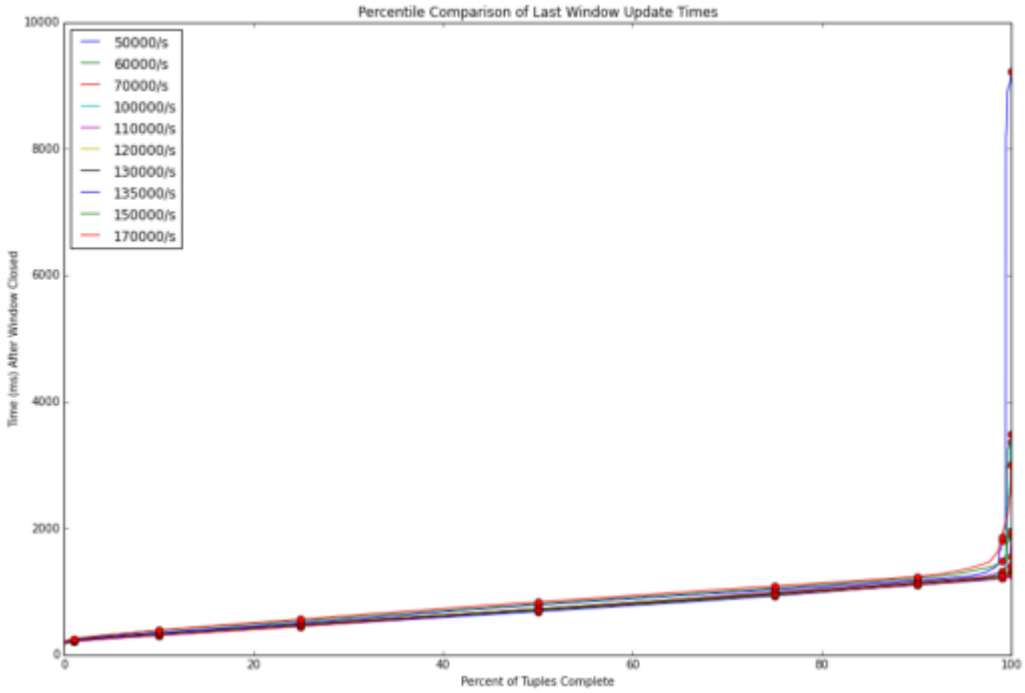


Figure 17. Flink Performance [35]

Apache Storm Benchmark - compared to Flink benchmark, 2 versions of Storm are tested. Storm 0.10.0 was not able to handle throughputs above 135,000 events per second. With acking enabled, Storm performed badly at 150K/s, with acking disabled, Storm even beat Flink for latency at high throughput.

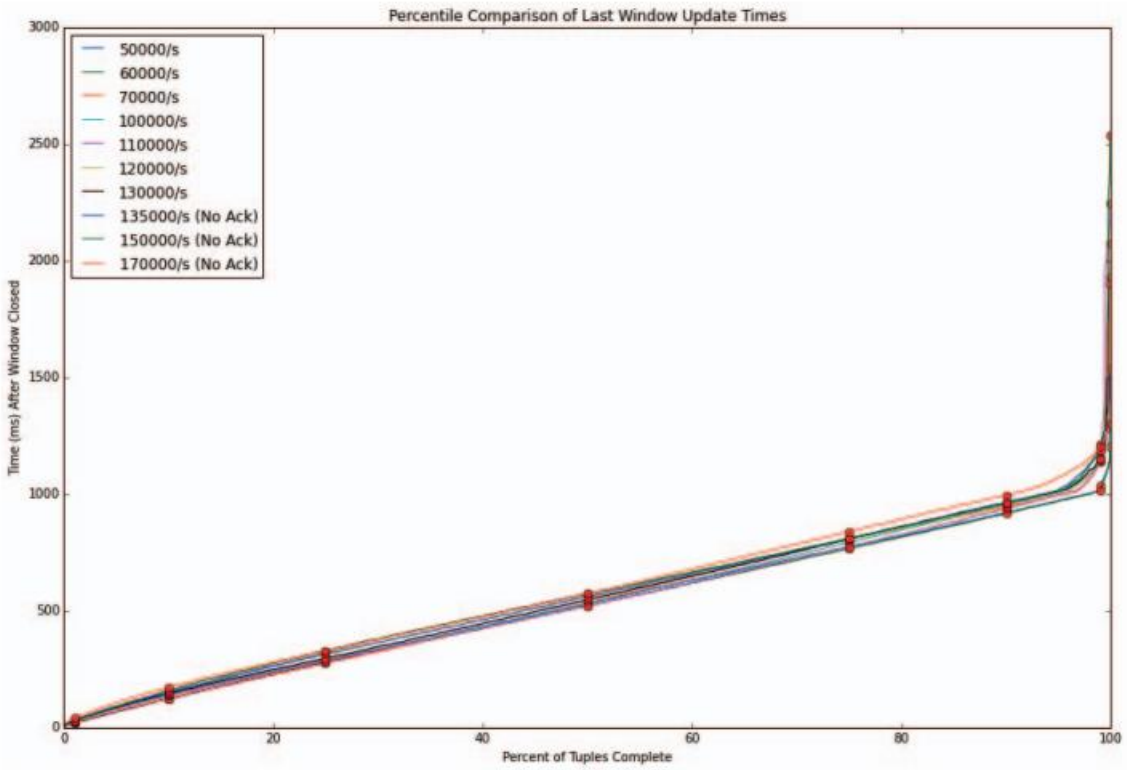


Figure 18: Storm Performance 0.11.0 version [35]

Apache Spark Benchmark - compared to others, it is written in Scala with DStreams.It was benchmarked within 3 and 10 seconds frames for updating Redis.Event consuming rate was less than others, which was 100k events/sec. Three interesting behaviors of Spark was observed during the experiment depending on window duration. They set batch duration to 10 secs, in which 90% of events have been processed. Majority of events was handled in the current micro batch if the batch duration is larger. Figure 16 illustrates that, percentile processing for 10 seconds batch duration. Within 3 secs, they got better results, which is described in figure 19, which is second interesting behavior that, reducing batch duration, events are processed in 3 or 4 subsequent batches. Reducing batch duration, Spark Streaming falls behind.

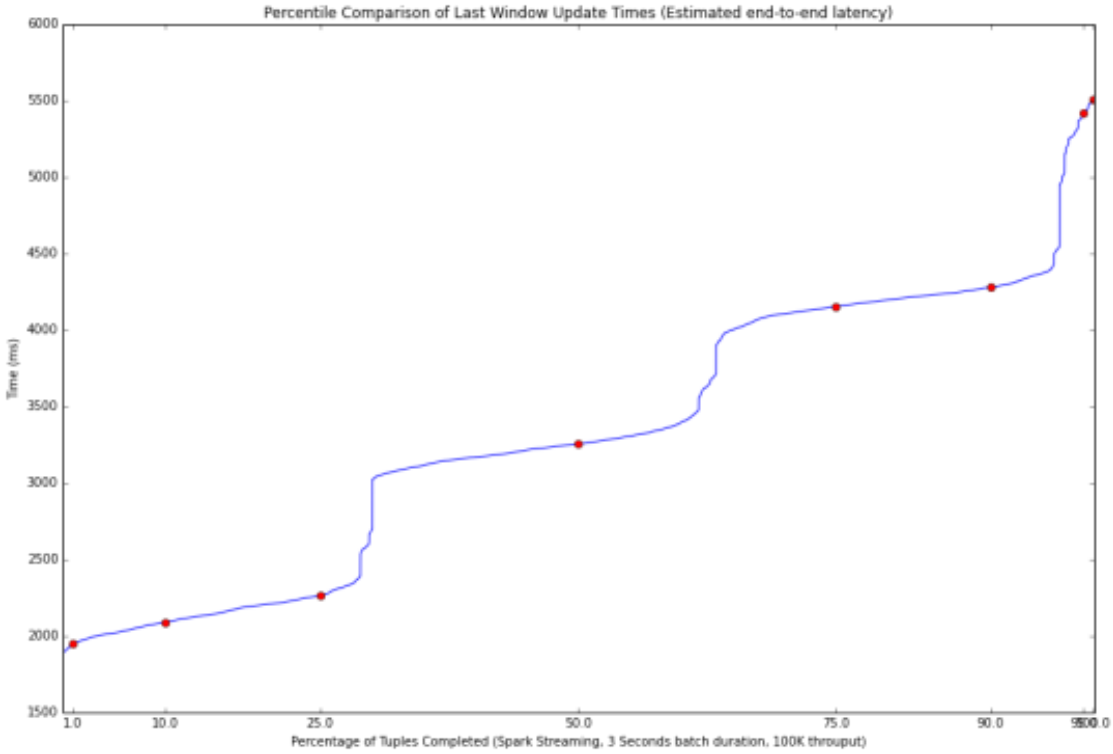


Figure 19. 3 seconds batch duration Spark Performance [35]

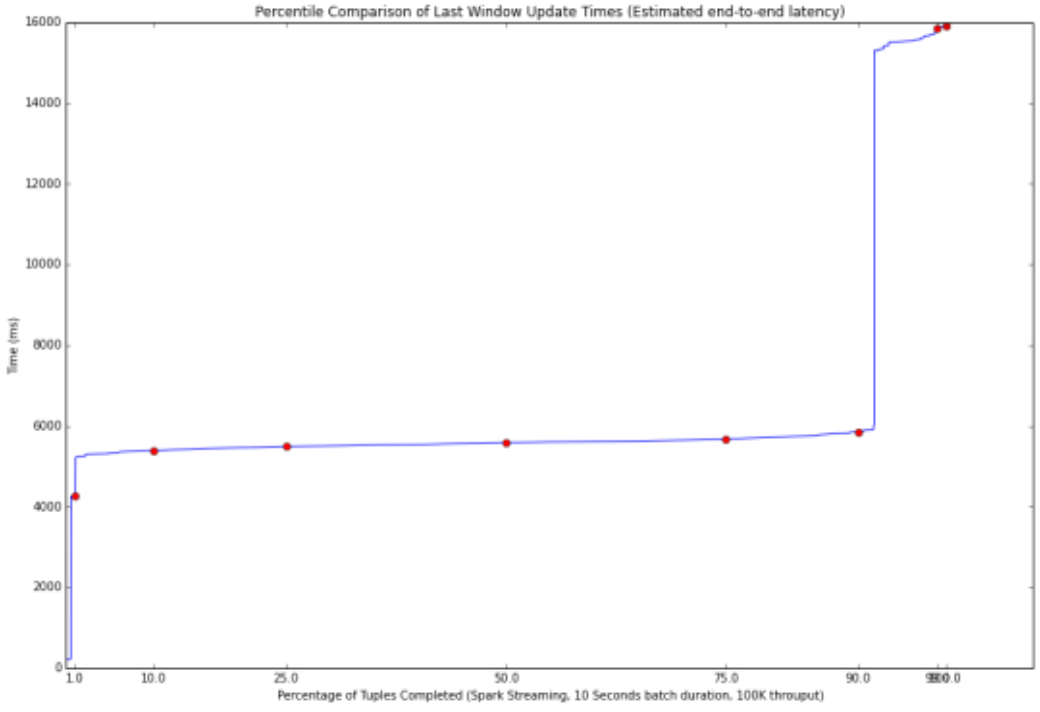


Figure 20. 10 seconds batch duration Spark Performance [35]

As a result of those benchmarks, we can see from the figure below, Flink and Storm both respond to incoming data as it is available, while Spark in a small time, regarding its micro-batch design. From the figure 22, Flink and Storm has similar performances, even with 0.11.0 version of Storm with acking disabled, beats Flink. while Spark has higher latency, it is expected to perform better throughput performance by configuring its batch interval larger. In conclusion of this work, for the cases, where real-time is the most important factor and there are no resource limitations, Flink can be chosen, for continuous processing and for high throughput, Spark Streaming is a good choice.

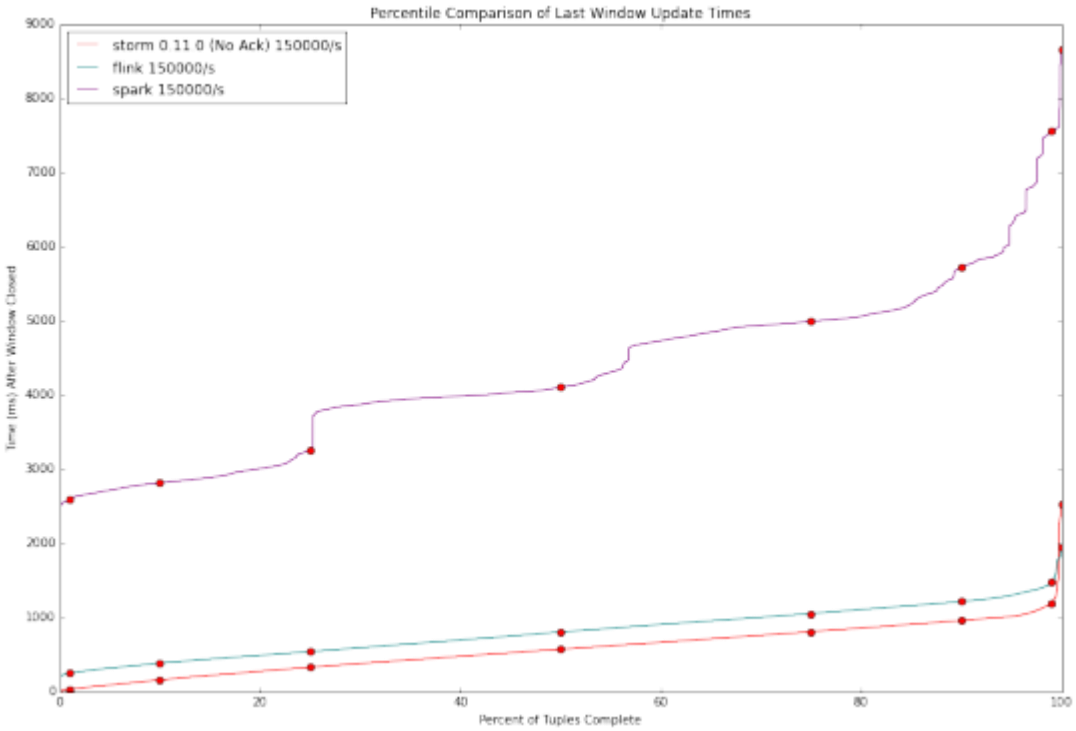


Figure 21. Performance Comparisons of Storm, Flink and Spark Streaming in term of window times [35]

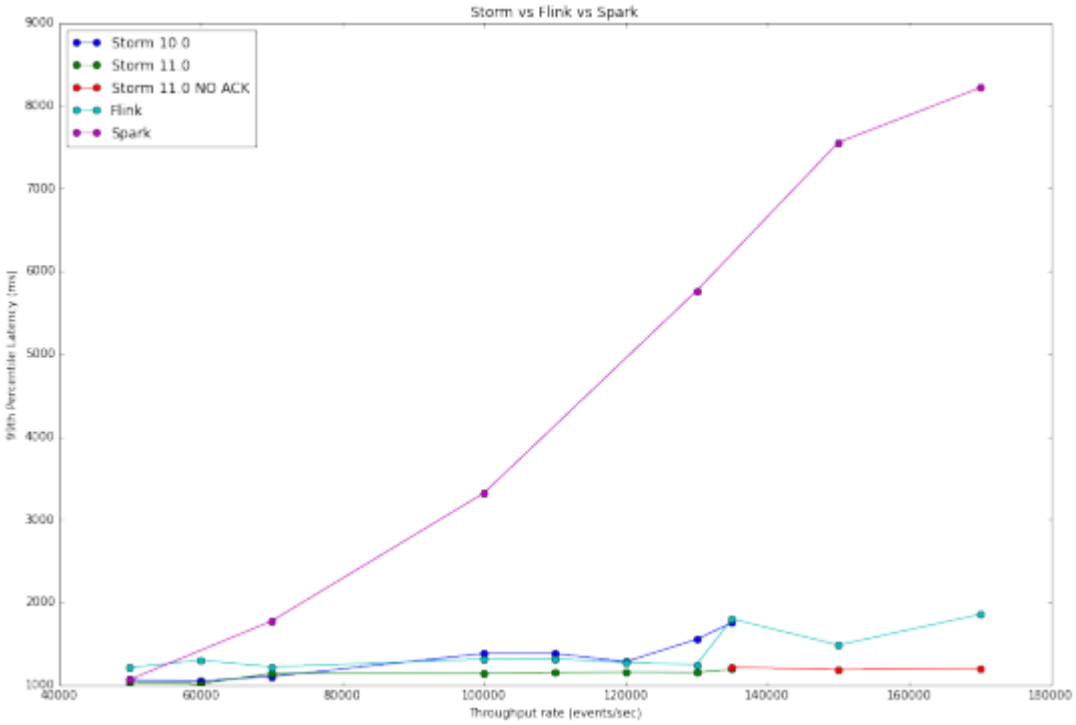


Figure 22. Performance Comparisons of Storm, Flink and Spark Streaming in terms of 99th percentile latency [34]

4.2 StreamBench

It is another benchmark for performance comparison with Flink, Spark and Storm by Yangjun Wang [36]. It is a java application, which consumes data from Kafka and processes data on clusters.

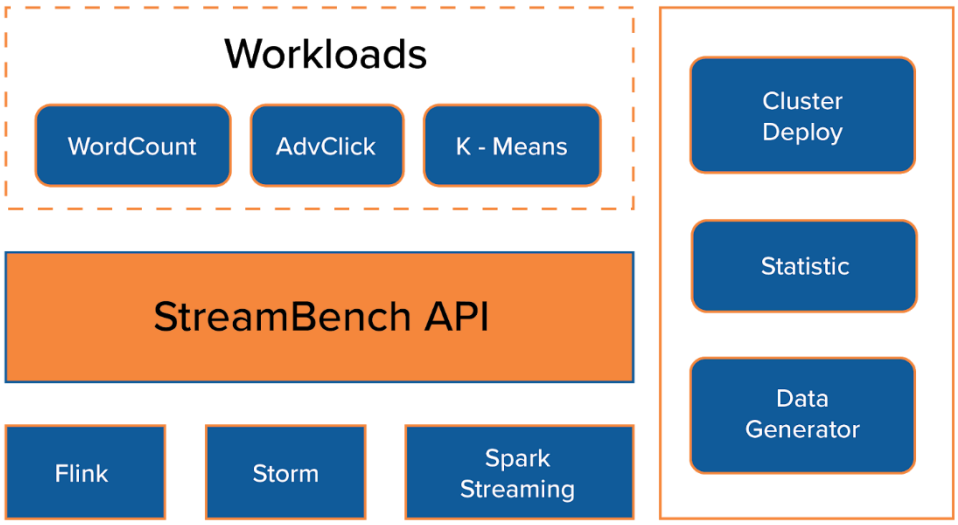


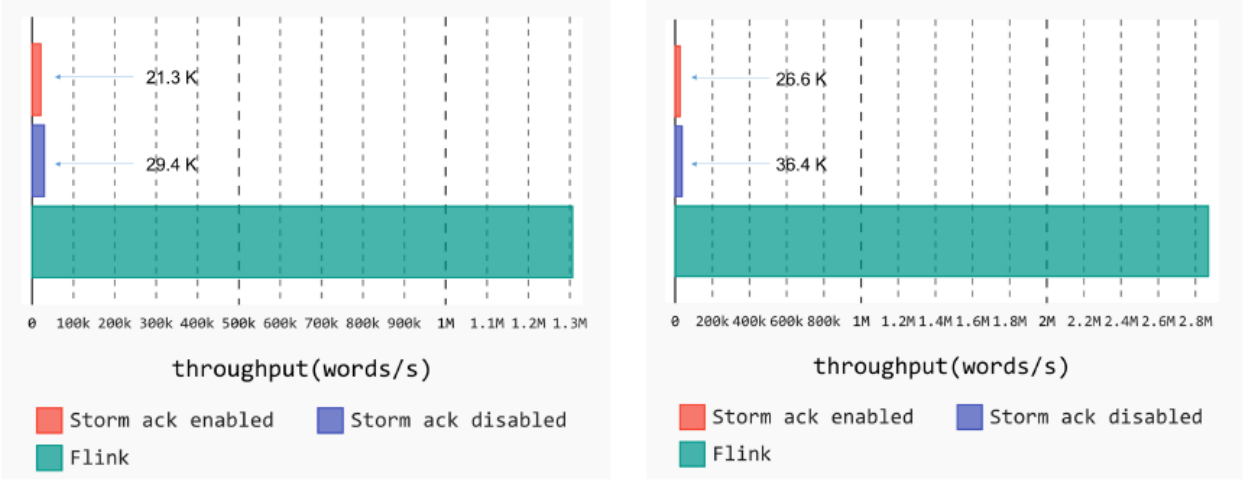
Figure 23. Architecture of StreamBench [36]

Versions of processing engines were following Spark-1.5.1, Flink-0.10.1 and Storm-0-10.0. With 2 clusters, a computer cluster which consists of 8 worker nodes and one master node, Kafka cluster which has 5 brokers.

StreamBench has 3 workloads inside as we see from the picture. Each workload has its own streaming application and one or more Kafka topics. Kafka cluster feeds application with messages and application does some basic operations.

WordCount is one of the workloads. Data generator generates data for the workload and it counts the number of words which appear in the input set. He implemented WordCount experiments in 2 different ways. Offline model, for throughput measure, Online model is used for latency. Apart from that, another version of WordCount which is named Windowed WordCount implemented regarding to bottleneck of the computation node. Within Windowed WordCount, the results which are calculated before, are kept in calculation nodes, when windows are closed, it calculates the final results.

In the Offline model, because of Spark's computation model, he only compares Flink and Storm.

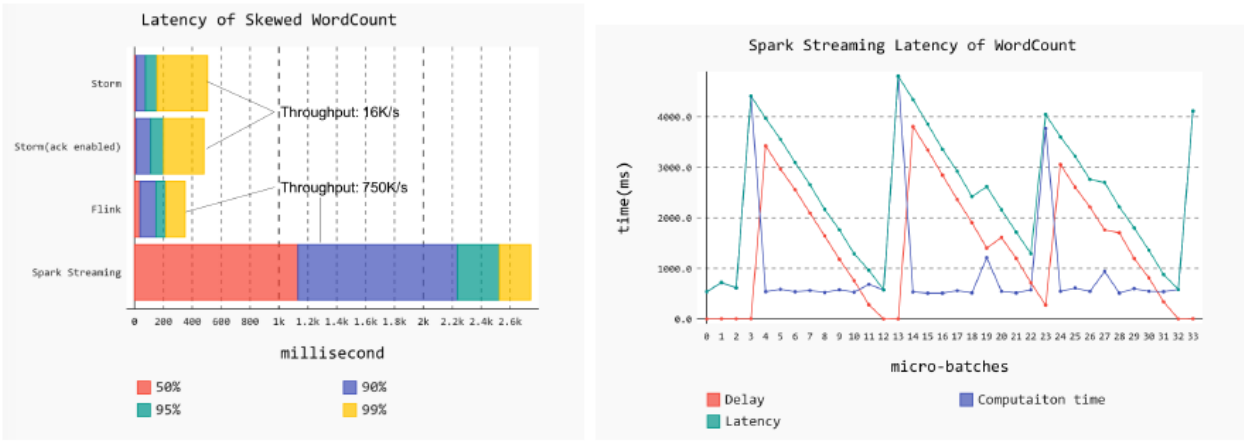


(a) Skewed WordCount (b) Uniform WordCount

Figure 24. Throughput of Offline WordCount (words/second) [36]

As we can see from the Figure 24, Flink performs 10 times better than Storm with uniform data. After changing the number of nodes from 4 to 8, performance of Storm within skewed data increases 58%.

On Online WordCount, he mentions Spark's default configurations for micro-batch and checkpoint intervals which are 1 and 10 seconds correspondingly, delays the application. Due to Spark's fault-tolerance, it tries to write information to the storage in each 10 seconds.



(a) Records latency (b) Micro-batches latency

Figure 25. Latency of Online Word Count [36]

Result for Storm is 10 milliseconds, where Flink has 39 milliseconds latency. Spark's 95th percentile latency was 217, where Flink's was 201.

Another workload is AdvClick, which serves for view and click of advertisement events. Each set is described as id and shown time. Yangjun Wang implemented join of clicked advertisements using id. As ids are appeared in both streams, is counted as a valid click. In this experiment,[36] he define 20 seconds for view event, where 5 seconds for window time for advertisement clicks which will be joined to other stream.

	Maximum Throughput	Latency		
		Throughout	Median	90%
Storm (ack disabled)	8.4K/s	4.2K/s	14ms	2116ms
Flink	63K/s	33K/s	230ms	637ms
Spark Streaming (20s/5s)	< 2K/s	Ø	Ø	Ø
Spark Streaming (60s/30s)	20K/s	20K/s	~20s	~24s

Figure 26. Advertisement Click Performance [36]

Flink had higher throughput and lower latency than others. Because of the interval time, Spark performs slower than others due to its micro-batch model.

Another experiment was KMeans, in which Spark achieves the best maximum throughput which is better than Flink and Storm.

From the conclusion of these related workloads, Flink performs better than others regarding throughput and latency at the same time. If low latency is needed then Storm and Flink are the best choice due to chosen workload related cases, for high throughput Spark is a good choice.

5.Contribution

In this chapter, I will talk about advantages, disadvantages of frameworks in micro-batch and real-time data processing.

In stream processing there are several important characteristics, which each engine should have.

Fault tolerance: One of the important characteristic, when there is a node failure, network issue, engine should be recover its state and start from the point where it was before.

Delivery Guarantees: incoming data should be processed although there is a failure. It can be guaranteed with the help of delivery guarantees.3 the most famous ones are available.

- **At-most-once:** The producer sends each message at most once and it doesn't need any responses. Because of that, this semantic is easy to implement. Disadvantage of this semantic is that there is no way to prevent message loss.

- **At-least-once:** The producer sends each message, until it gets a response from the receiver, otherwise it will send until it gets the response. This semantic guarantees that each message has been received. Disadvantage of this semantic is that duplications may occur, because of each message might be delivered multiple times in a row. This semantic is used by Facebook [37] and Twitter [38].

- **Exactly-once:** This semantic guarantees that each message will be delivered exactly one time, not less or never. Among other semantics, this one is the best. Disadvantage of this semantic is that it requires additional resources [39].

At least-once will be processed at least one time even in case of failures, at most-once may not be processed in case of failures or Exactly-once will be processed one and exactly one time even in case of failures.

Performance: 2 main things needed to be taking into consideration. Latency, Throughput.

Advanced Features: Some of the engines don't provide some functions such as Windowing, Event Time Processing and etc. In Complex Event Processing, these features are so handy.

	Apache Kafka	Apache Flink	Spark Streaming	Apache Storm
Fault tolerancy	Yes	Yes	Yes	Yes
Delivery Guarantee	At least once	**Exactly once**	At least once	At least once
Performance	**Low Latency/High Throughput**	**Low Latency/High Throughput**	High Latency/High Throughput	Low Latency/Low Throughput

Table 2. Comparison of characteristics

5.1 Advantages and disadvantages of streaming engines.

Advantages of Kafka: Simple APIs, it has a very light weight library, you don't need a dedicated cluster, supports stream joins, implicit support with RocksDb, good for micro services, ETL, IOT applications. Kafka is used more than 1000 companies in the world [12].

No need to setup a separate cluster, therefore no extra operational burden. Kafka Streams can be embedded into the existing Java-based application or micro-services just like any other dependencies. Existing standard CI/CD pipelines for deploying micro-services can be used for deploying Kafka Streams based stream processing services. Kafka Streams DSL is perfectly capable for handling our stream processing use cases which can be quickly prototyped locally

Disadvantages of Kafka: Doesn't support watermarking, doesn't support wildcard for topic selection, hard to maintenance Kafka cluster, scaling is hard, auto-scaling is harder, queues increase performance decreases.

Advantages of Flink: Native support of batch, real-time, back pressure can be handled from system architecture, ML and etc., high performance, efficient memory management, supports Lambda Architecture, less configurations, reusing the code which is written for Storm.

Disadvantages of Flink: High CPU usage, community is not wide, uses raw byte for internal data representation.

Advantages of Spark: Supports micro-batch streaming, APIs are easy to use, faster than Hadoop.

Disadvantages of Spark: Automatic optimization is not possible, not owning file management system, high latency, weak back-pressure handling, consumes a lot of memory.

Advantages of Storm: Supports real-time streaming, low latency, with Trident can be used for micro-batches.

Disadvantages of Storm: As itself doesn't guarantee order of messages, no time and session window, it is stateless.

As I already mentioned in the previous chapters about real-time and micro-batch streaming, in the table below we can see benefits of these 2 models.

	Real-time	**Micro-batch**
Latency	**Lower**	**Higher**
Throughput	**Lower**	**Higher**
At-least once	**Supports**	**Supports**
Exactly once	**Supports**	**Supports**
Implementation	**Easy**	**Easy**

Table 3. Real time vs micro-batch comparison

6. Experiment

6.1 Experiment design.

In this section, I will provide information about the analysis that I made with streaming engines, to compare their performance on latency, throughput and find the suitable one for the experiment. Environment is used for testing has 3.60 GHZ Intel® Xeon(E) CPU with 16 cores and 128GB RAM with network bandwidth 1Gb/s and single node.

This experiment is a real industrial case, which I faced at my workplace, built with Kafka, Apache Spark and Apache Flink for consuming randomly created securities, and calculating occurrences (sell and buy amount) of these securities from Kafka and implement MapReduce algorithm on them to find the most famous one based on their transaction amount regarding to securities' symbol. After finding the most famous one from the data that fed to the system, famous ones with highest volume and other merged important data are sent to search engine such as Apache Lucene for character based searching.

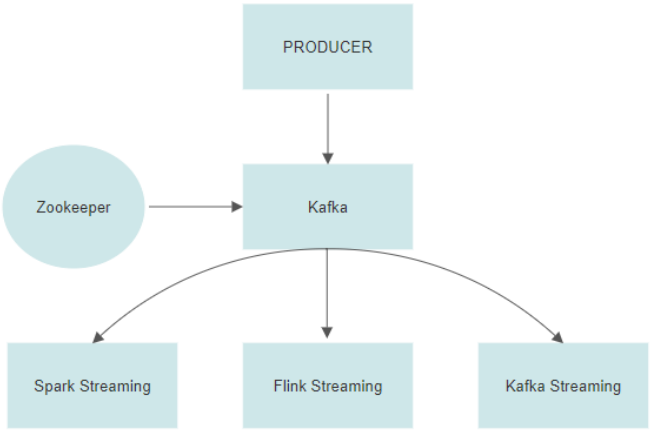


Figure 27. Experiment design

For applying micro-batching, with spark batch interval was changed between 100 ms to 5 seconds, where the random elements which are created in the last 100 ms, becomes a single batch for further processing. I used mapReduce [40] model for this experiment. In the created batch, mapping elements into tuples as <word,occurences>, then reducing the tuples by element and sum of all occurrences. I repeated each batch process for a few times 5 minutes long to get average processing time and input records.

With Flink and KStreams, I repeated the same process, using Flink's DataStream API and KStreams for testing real-time streaming.

In the experiment, approximately 10K securities' symbols from Nasdaq are fed into Kafka. With the help of Kafka Producer, randomly chosen symbols are sent to Kafka topic within 100ms time interval. As securities symbols are strings, and values also can be defined as string, StringSerializer class is used. For creating Kafka topic, Zookeeper Client is used for establishing connection with Zookeeper Server, then topic is created with only one replica and partition. Each engine was configured to get messages from the topic and implemented basic operators on them to find most famous stock. Most of the configurations of these frameworks left at their defaults. Micro-batch size tested in different interval, for getting better overview of the performance, I decided to share the results within 3 seconds interval. All of the metrics are gathered by frameworks' in-built metric tracking functionalities. For example, Spark provides web user interface for streaming and structured streaming. With the help of Gauge, Counter and Meter interfaces, it is possible to access these metrics in Flink [41].

6.2 Spark's results.

For loading the engine with more data, during this experiment 10 to 25 Kafka producers produced between 10K and 120K records per second input data. Another approach for generating more data was adding more threads, that use the same producer. But thanks to Docker, with a line of command, it is possible to run many producers at the same time. I configured Spark to get messages from Kafka topic. Batch duration was set to 3 seconds, after reading messages from Kafka, mapReduce programming model is used for processing input data. Spark gets input stream of symbols for the last 3 seconds, map them into tuples <symbol, amount>, reduce tuples by symbols and sums the amount of them. Figure 28 describes the DAG visualization of the spark job for this experiment.

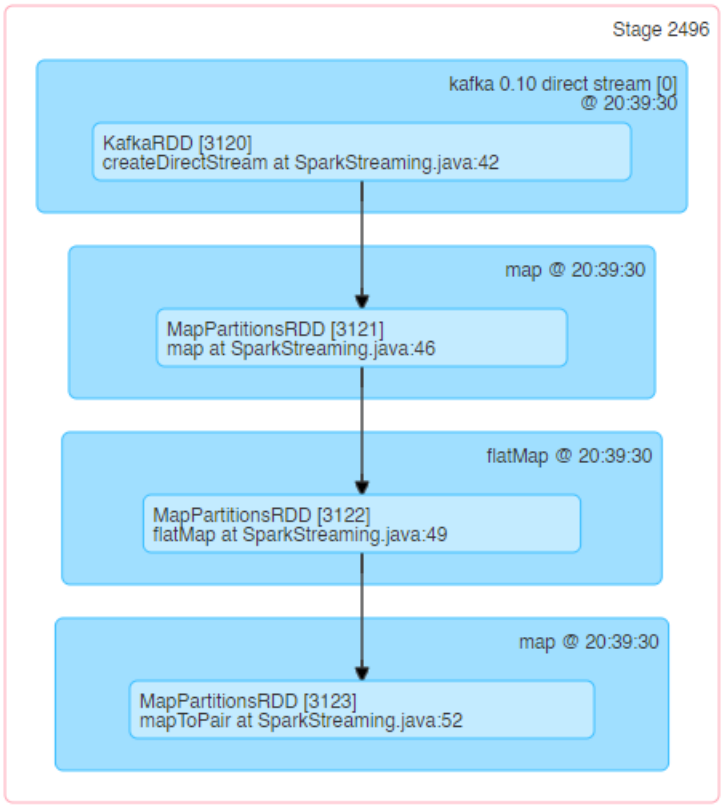


Figure 28. DAG vizualition of the Spark Job.

Compare to other related works, latency in this thesis, could be lower. Because, latency is calculated from the time when the event is inserted to engine. Another approach is adding timestamp for input and output data, which consumes few ms during these processes. For this experiment, micro-batch is set to 3 seconds for both DStream and Structured Streaming. Experiment was repeated many times within 5 to 20 minutes' frame.

Spark Streaming: In the figure 29, average input rate is 34125 records per second. Each batch consists of more than 100K records. Figure 30 shows average processing time latency. Average throughput and latency in this case is 53K records/sec and 2 secs respectively. As processing time less than batch interval, Spark Streaming performs quite well in this case.

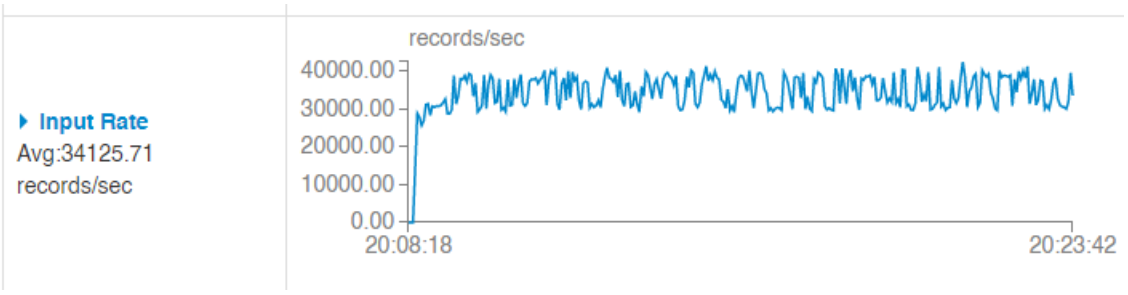


Figure 29. Input rate for Spark Streaming.

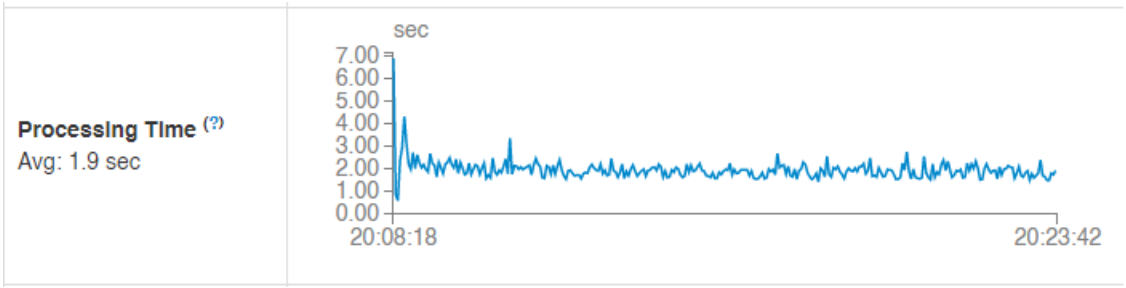


Figure 30. Processing time for Spark Streaming.

Spark Structured Streaming: The same experiment is repeated for the Spark Structured Streaming. The best result for process rate and batch duration perspective for the structured streaming is displayed in the graphs 32 and 33. Kafka produced approximately 120K records per second, Structured Streaming was able to process approximately 105K records per second which is higher throughput than Spark Streaming. Latency for each batch is about 10 seconds, which is illustrated on Figure 33, which is not good when micro-batch time is 3 seconds.

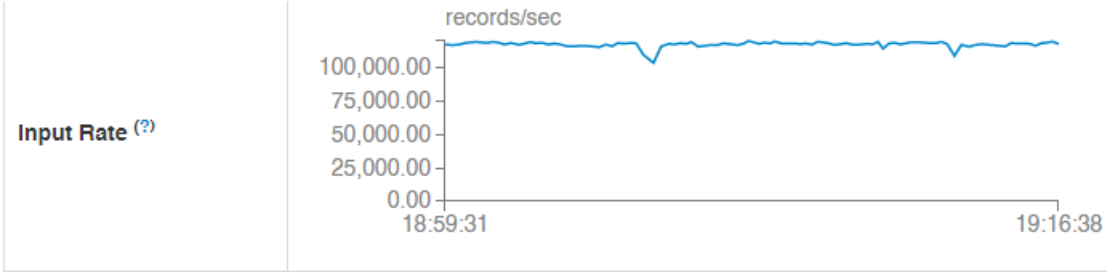


Figure 31. Input rate for Spark Structured Streaming.

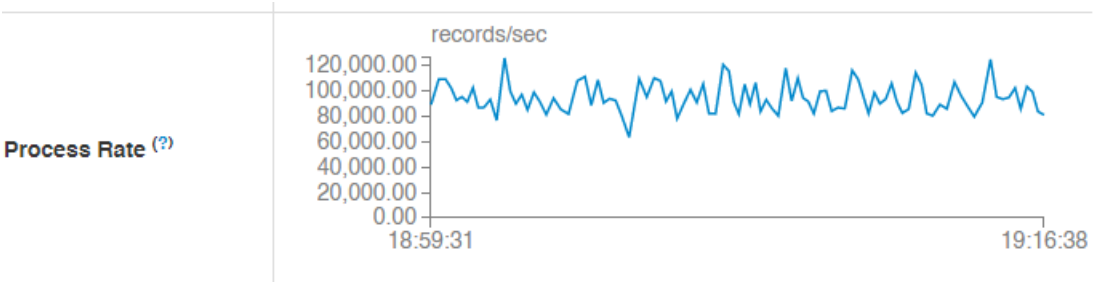


Figure 32. Processing rate for Spark Structured Streaming.

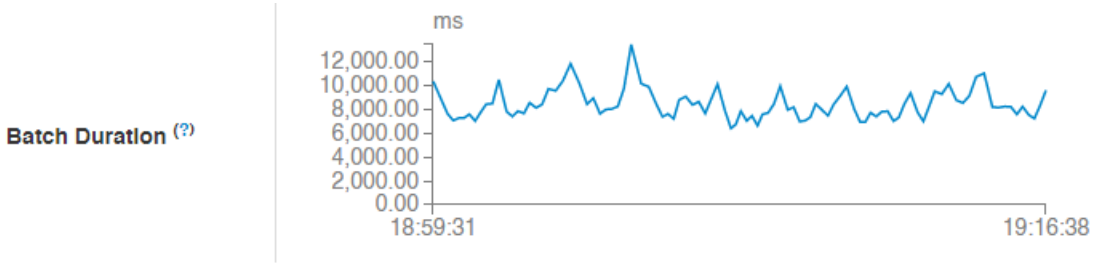


Figure 33. Batch duration of Spark Structured Streaming.

In this experiment, while increasing input size Spark Streaming's latency increasing significantly. This experiment, shows that with low latency, Spark Streaming is capable to provide medium throughput, if high throughput is the first priority, then Spark Structured Streaming can provide it, when latency is in the second priority. Figure 34 illustrates latency comparison of Spark Streaming and Structured Streaming while micro-batch interval set to 3 seconds and input rate is changing on x-axis from 50K to 150K.

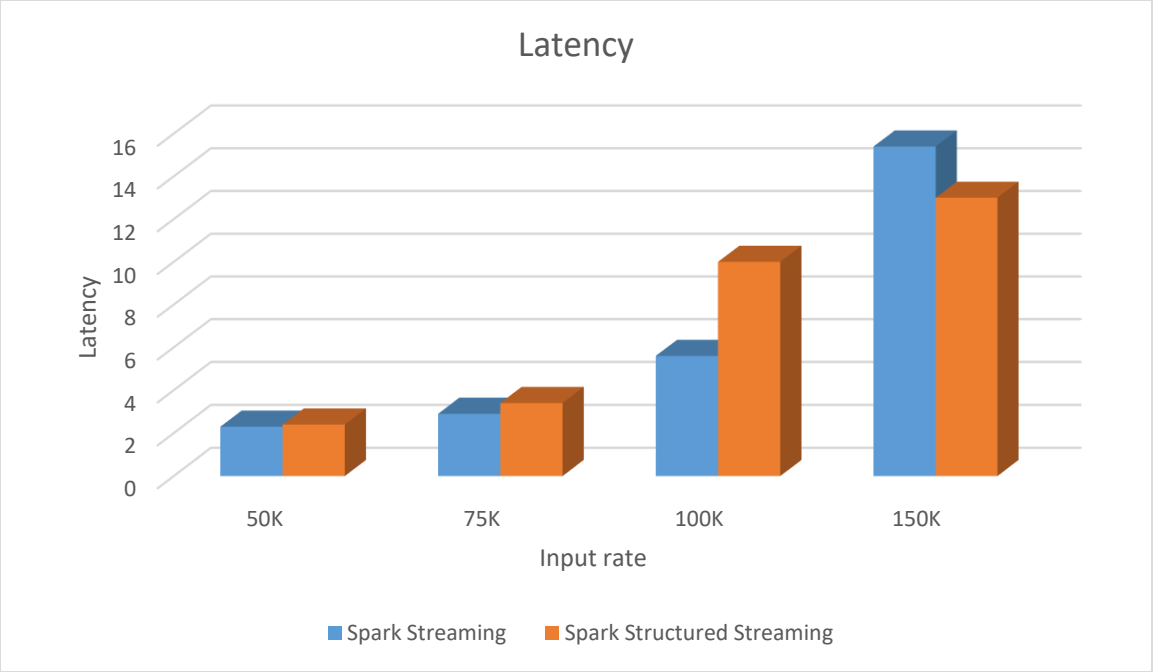


Figure 34. Latency comparison of Spark Streaming and Structured Streaming.

6.3 Flink's results.

The same experiment is repeated for the Flink in the real-time without any batch interval. Input rate was between 30K to 200K for this experiment. Checkpoint was not enabled because of time consumption. Flink outperformed great performance within high throughput as it processes message as soon as it receives due to its time series based approach. In the figure 35, illustrates experiment of the Flink job.

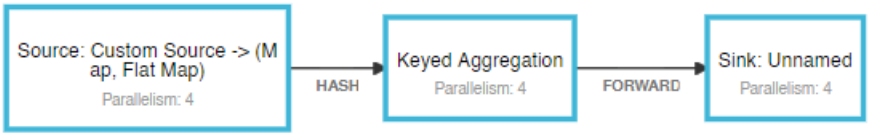


Figure 35. Flink job.

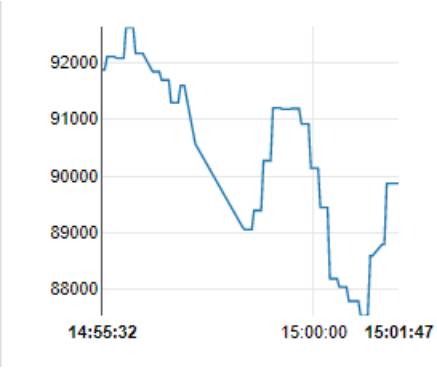


Figure 36. Flink throughput graph

Lowest latency is gained from this experiment for Flink was around 600ms which is quite good, average latency was changed between 800 ms to 1.2 seconds depending on the data intensity for 80 to 120K records.

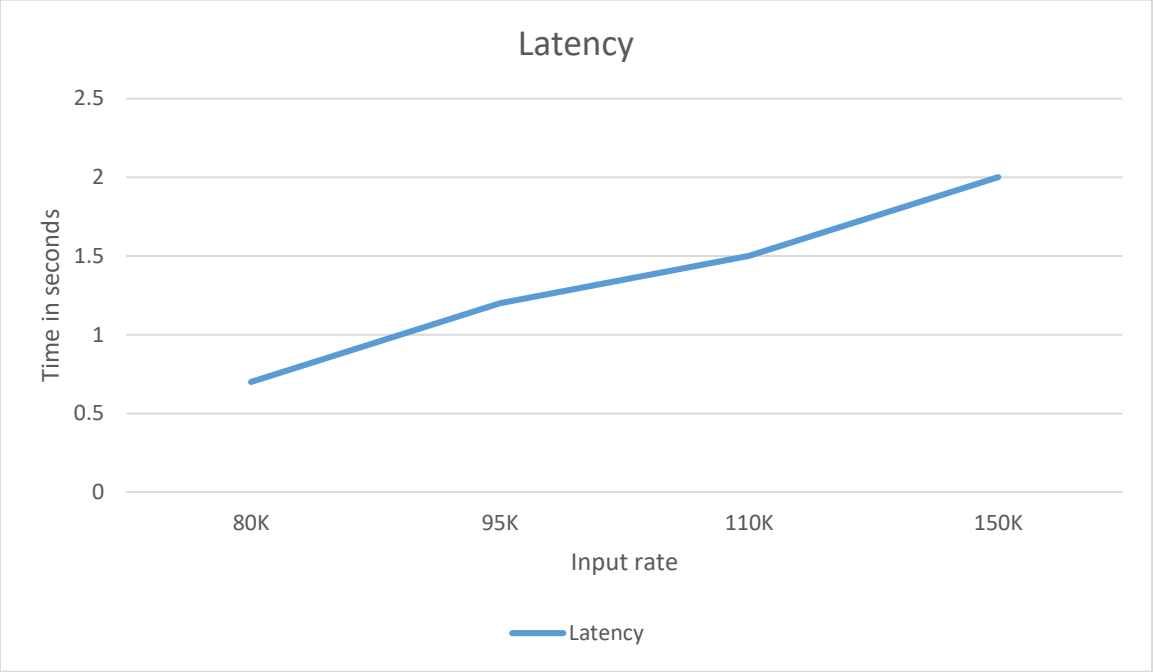


Figure 37. Flink latency based on data intensity.

Another experiment has done with Flink within 3 seconds time window. Figure 38 illustrates Flink job. As previous experiments, input rate by Kafka was changed 50K records/sec to 150K records/sec for this experiment. The highest processing record was 263 M during 16 minutes, in this experiment, average latency was 2.3 seconds, with approximately average 67K records/sec

throughput. Increasing input rate caused the latency increasing linearly until 100K. After 100 K latency was more than 6 seconds. Figure 39 illustrates latency changes by input rate.

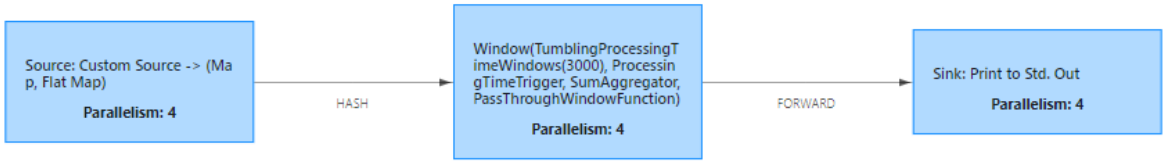


Figure 38. Flink 3 seconds time window

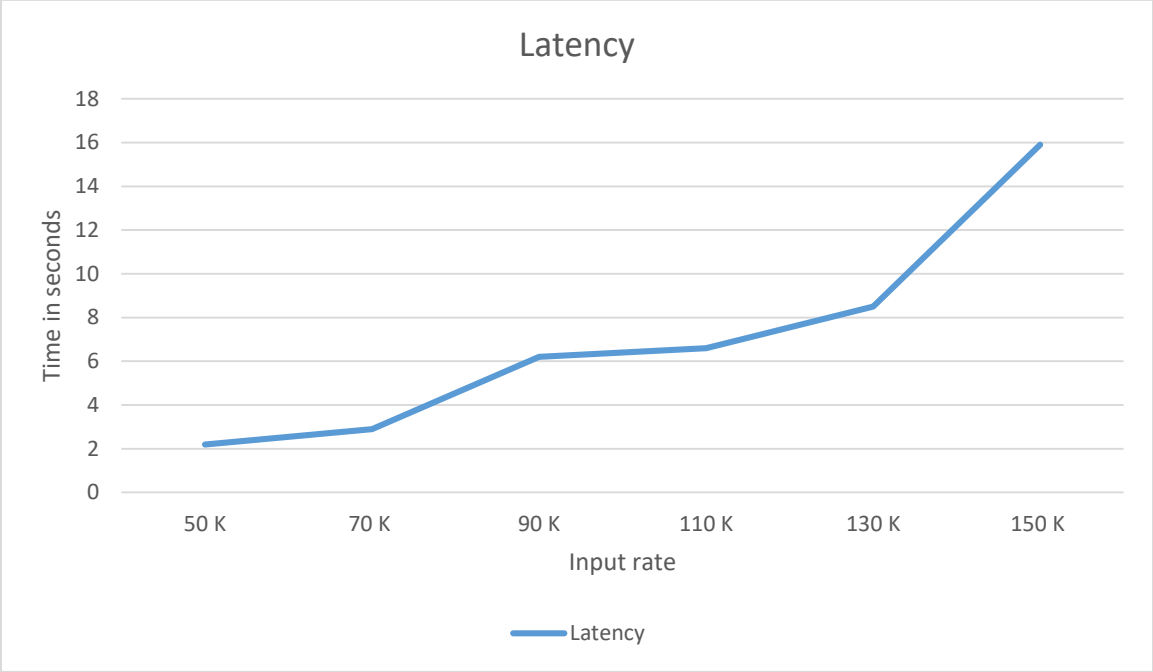


Figure 39. Latency of Flink based on input rate

6.4 Kafka Stream's result.

Last experiment of this chapter is done with Kafka Stream. The same logic is implemented for Kafka Stream too. Data ingestion rate changed between 30K to 200K. Kafka Streams were capable of handling until 120K records per second, within linearly increasing latency which was changing between about 900ms to 4 seconds. After 120K records, latency was increasing rapidly, CPU and memory usage wasn't normal. Average latency for running 15 minutes with input rate 120K records/sec was 3.5 seconds. Highest throughput was 90K/sec with the lowest latency 2.1 seconds.

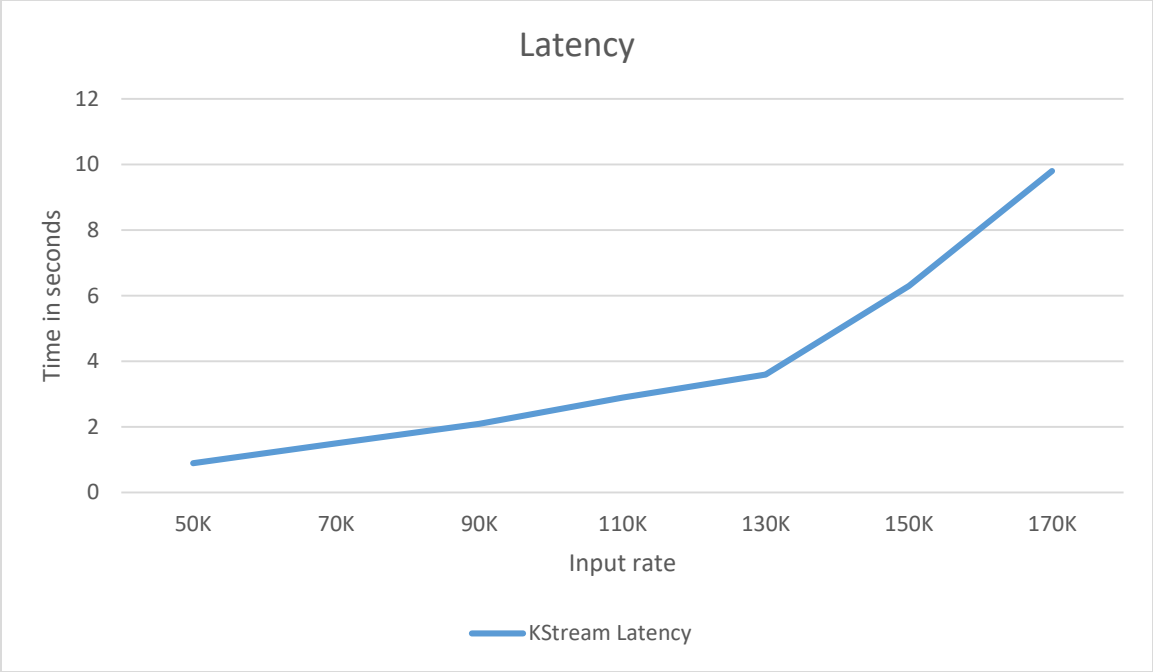


Figure 40. Kafka Streams Latency

In conclusion of this experiment, from performance perspective in real-time, Flink has a good throughput and low latency compare to Kafka Stream, which is slightly differ in throughput, however in latency, bottlenecks start after 120K records for Kafka Stream. Figure 41 illustrates latency comparison of Flink and Kafka Streams for real-time.

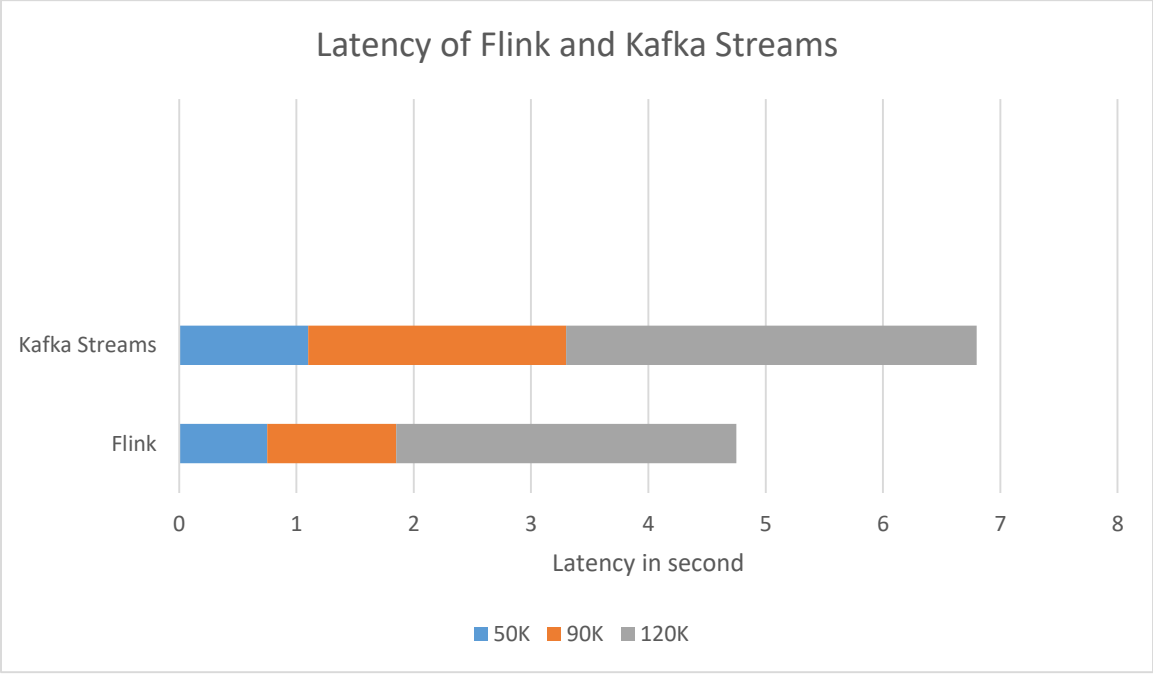


Figure 41. Comparison of Flink and Kafka Streams in real-time.

I have already mentioned Spark's and Flink's performance in the previous experiments regarding micro-batch size with 3 seconds. As micro-batch size was 3 seconds, Spark and Flink's latency is illustrated on Figure 42. From the figure, we can see that, Spark Streaming wasn't trustful after 100K and latency was increasing rapidly. Flink compare to Spark Structured Streaming performed quite well regarding to latency.

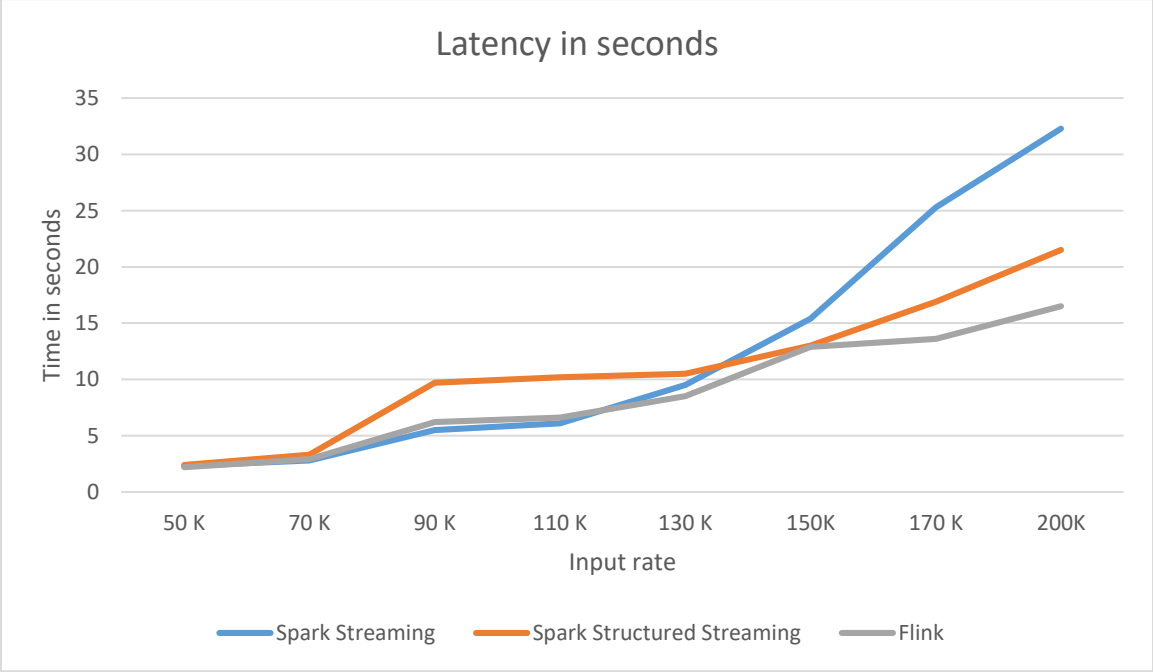


Figure 42. Latency comparison on micro-batching.

In the figure 43, average throughput comparison of these frameworks is illustrated with different micro-batch duration and input rate 100K records per second. As we can see from the charts, Spark Structured Streaming is a winner in this race because of its micro-batch stream processing model. However, from the previous figure, we can see that Spark achieves this with higher latency compare to Flink. As micro-batch duration is increasing processing capability of Spark Structured Streaming is also increasing. Flink also performed quite well, it was able to reach approximately 200K records/sec. Spark Streaming's throughput was 2 times lower than Flink's.

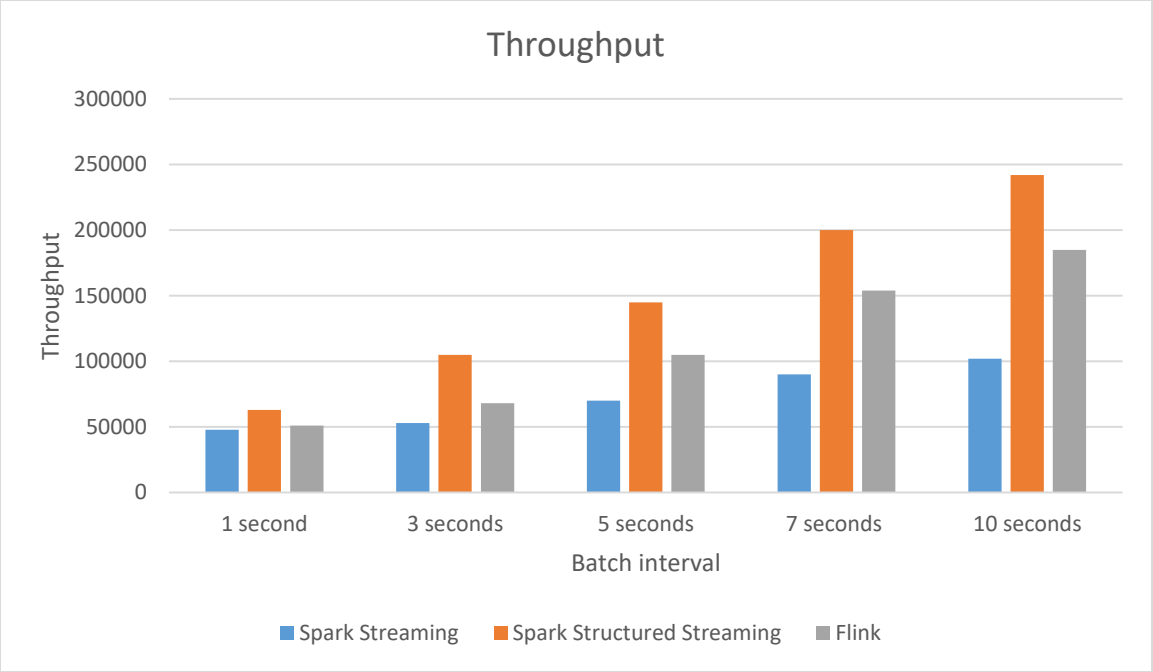


Figure 43. Throughput comparison of frameworks.

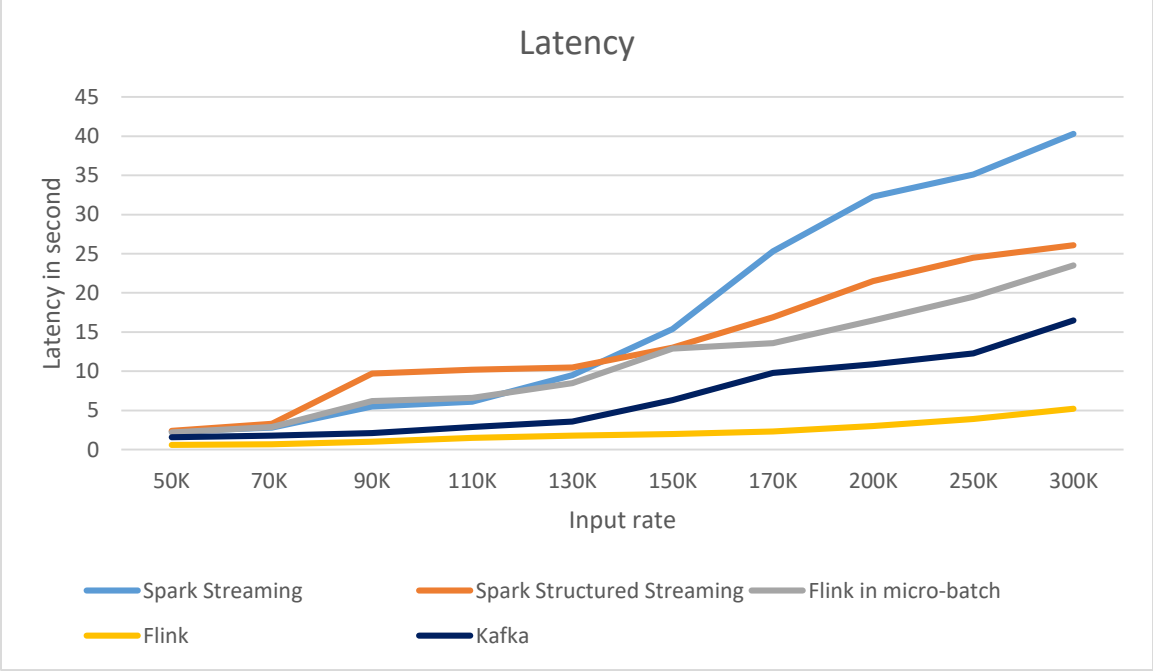


Figure 45. Latency comparison of experiment.

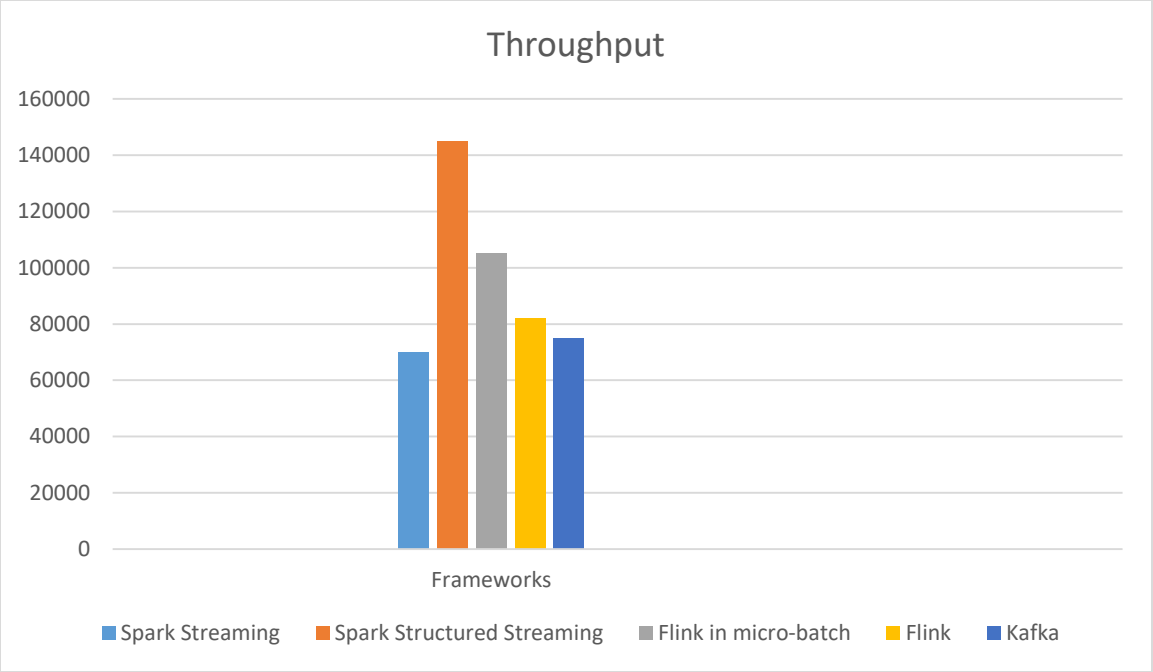


Figure 46. Average throughput comparison in real-time vs micro-batch.

In conclusion of this chapter, from the experiments, I would like to summarize that, for real-time stream data processing from figure 45 and 46, Flink and Kafka Streams performed quite well, Flink had advantages from both latency and throughput perspective. If throughput and latency are both important in grading, Flink is the most interesting choice. For micro-batching stream data processing Spark due to its engine design, showed great throughput compare to others. For higher input rate we saw that Spark Streaming didn't perform well. From this perspective, Flink can be used in both real-time and micro-batching. Even with some workarounds on tuning, throughput of Flink can be improved. Spark Structured Streaming due to its high throughput is a great match for micro-batch stream processing, when latency is not important and minimal tuning is required. Not also from performance side, but also, maturity of framework Spark is more mature and have larger community than others in streaming. For reasonable throughput and latency, when data is in Kafka cluster, Kafka Stream is also a good alternative to Flink. However, apart from the performance perspective, in real life scenario of this experiment, as data ingestion wasn't large, high throughput wasn't the first priority and maintaining other frameworks to run in cluster were costly, I chose Kafka Streams, which handles the scenario quite well.

6.5 Guidelines on frameworks.

After getting to know the default capabilities of the engines, in this part, I will share some of the methods for improving performance. Main action for Spark is setting the right batch interval so that batch data will be processed as soon as they are received. Data serialization is also an important factor on Spark tuning, by default Spark uses Java serialization, however Kryo serialization is 10x faster than Java serialization [42]. Other parameters may be very specific to the use case such as data size, processing flow etc. For example, input data streams retrieved from HDFS, S3, or file systems such as ext3, ext4 can also affect the performance. Configuring the correct buffer time out can increase the performance of the Flink for low latency requirements. Checkpoints should be reviewed carefully before the set up. While developing with Kafka Streams, depending on the use cases, high level stream DSL and Producer API shows different results in performance. In conclusion, there are no all-round winners, use cases are one of the important factors to mention, for choosing the right streaming framework. While choosing the right framework, factors should be pointed out are the following.

- How much data will you process? (Throughput)
- How fast should it be? (Latency)
- Who will build it and functionalities? (Programming languages, Level of APIs, functionalities such as windowing, joining, capability of using SQL)
- Do you need ordering? (event or processing time)
- Will it be batch integrated? (possibility of using batch API)
- Running environment (YARN, Mesos)
- How much state do you have? (delivery guarantees, checkpoints etc.)

Table 3 consists of the answers for the factors above. If high throughput is the first priority, then Spark is the best choice, if latency is the first priority such as in Fraud detection, then Flink is a good choice. In usage of programming languages, Spark beats other as it offers Scala, Java, Python. Flink from 1.9.0 supports Python, Kafka Stream only supports Java and Scala. All of these provide high level APIs and functionalities such as windowing, joining and using SQL are possible in all frameworks. Time characteristics such as processing time and event time are offered by all frameworks. Integration to batch, as using batch api is highly provided in Spark, limited in Flink and Kafka Stream doesn't offer batch integration. Spark and Flink can be run on MESOS, YARN cluster managers and also in standalone, where Kafka Stream runs on Kafka clusters. All of them are fault tolerant, support exactly-once processing. In general, Kafka Stream is mainly used for ETL processing, Spark is used for ETL and ML, Flink for Complex Event processing.

	Spark	Kafka Streams	Flink
Throughput	High	Low	High
Latency	High	Medium	Low
Execution model	Micro-batch	Continuous	Continuous
API	High level, SQL	High level, KSQL	High level
Guarantee	Exactly once	Exactly once	Exactly once
Streaming source	Socket, file system, message queues	**Kafka only**	Socket, file system, message queues

Table 4 – Comparison of frameworks

7 Future Work.

From the experiments and theoretical part of this thesis, I can summarize the frameworks that, in real-time scenarios, in which result should be available in a quick manner and resource availability is quite generous, Flink must be preferred than others. For micro-batch stream data processing, due to its streaming processing model, high throughput, ease of use, api level, functionalities such as ML, SQL, graph processing Spark is an adequate one.

For building better data pipeline, it is not only about choosing the right engine as in distributed systems, it is not an easy task to build the system which does the right things always. As I discussed in previous chapter **low latency and throughput**, are the main things to be taken into consideration for choosing right engine for real-time or micro-batch processing. Apart from these, api level of framework, language support, connectivity option for other data sources such as message queues, file systems and etc. are also important factors. **Scalability** is another key factor to maintain the performance while data loading or adding new resources. Apart from those, adding new features, new functionalities to the system should be easy and in minimal cost, which means **extensibility** of pipeline for your future work. In conclusion, I would like to summarize that, easy usage of Flink APIs, latency in real-time and adequate throughput rate make Flink the best choice for the experiment I have done for this master thesis. However, we have to keep in mind that, not every framework will be silver bullet for other cases. Each of them has their own pros and cons. As this thesis covered some well-known frameworks, however there are many other stream processing frameworks which hasn't covered in this thesis, which is in daily use. One of them is Storm which was intended to make an experiment also in this thesis apart from theoretical part. Other frameworks such as Apache Samza, Apache Beam, Apache Flume, Apache Kinesis Streams, Apache Apex would be interesting to make experiments on them to observe their behavior on real-time or micro-batch streaming.

As a part of future work, monitoring network usage, resource consumption, covering unit tests for applications would be nice to implement.

8. References

[1] The world beyond batch streaming. [Online]. Available: https://www.oreilly.com/radar/the-world-beyond-batch-streaming-101/

[2] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz Facebook, Inc. “Realtime Data Processing at Facebook”,2018. [Online]. Available: https://research.fb.com/wp-content/uploads/2016/11/realtime_data_processing_at_facebook.pdf

[3] Micro-batch-processing, [Online]. Available: https://hazelcast.com/glossary/micro-batch-processing/ (12.05.2021)

[4] 4Vs of Big Data. [Online]. Availabe: https://www.bigdataframework.org/four-vs-of-big-data/ (12.05.2021)

[5] Weise Thomas, Ramanath V. Munagala, Yan David, Knowles Kenneth, “Learning Apache Apex”,2017, p. 41-43.

[6] Ben Stepford,” Designing event driven systems”,2018, p. 52.

[7] Lambda Architecture. [Online]. Available: http://lambda-architecture.net/

[8] Kappa Architecture. [Online]. Available: https://infa.media/3y5RgNX

[9] Comparison of Lambda and Kappa. [Online]. Available: https://luminousmen.com/post/modern-big-data-architectures-lambda-kappa/

[10] Ivan Kovačević, Igor Mekterović, “Novel BI data architectures”,2018. [Online]. Available: https://www.researchgate.net/figure/The-kappa-architecture_fig5_326704892

[11] Kappa architecture. [Online]. Available: https://hazelcast.com/glossary/kappa-architecture/

[12] Introduction to Kafka. [Online]. Available: https://kafka.apache.org/intro

[13] Kafka Documentation. [Online]. Available: https://kafka.apache.org/documentation/

[14] Kafka Record Flow. [Online]. Available: https://www.cloudkarafka.com/blog/part1-kafka-for-beginners-what-is-apache-kafka.html

[15] Kafka Api. [Online]. Available: https://kafka.apache.org/documentation/#api

[16] Kafka Producer. [Online]. Available: https://bit.ly/3w0gFa7

[17] Kafka developers guide. [Online]. Available: https://kafka.apache.org/20/documentation/streams/developer-guide/config-streams.html

[18] Kafka Streams. [Online]. Available: https://kafka.apache.org/documentation/streams/

[19] Ben Stepford,” Designing event driven systems”,2018, p. 9.

[20] Apache Flink Documentation. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.13/

[21] Apache Flink. [Online]. Available: https://data-flair.training/blogs/apache-flink-big-data-unified-platform/

[22] Apache Flink Dataset. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/dev/dataset/overview/

[23] Apache Flink API. [Online]. Available: http://robertmetzger.de/incubator-flink-website/docs/0.5/java_api_guide.html

[24] Apache Flink Table. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/dev/table/overview/

[25] Apache Flink CEP. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/libs/cep/

[26] Stream Processing with Flink. [Online]. Available: https://www.slideshare.net/InfoQ/stream-processing-with-apache-flink

[27] Apache Spark. [Online]. Available: https://spark.apache.org/

[28] Apache Spark Architecture. [Online]. Available: https://dwgeek.com/apache-spark-architecture-design-and-overview.html/

[29] Apache Spark Documentation. [Online]. Available: https://spark.apache.org/docs/latest/

[30] Apache Spark Guide. [Online]. Available: https://spark.apache.org/docs/latest/streaming-programming-guide.html

[31] Apache Spark Performance Tuning. [Online]. Available: https://spark.apache.org/docs/latest/streaming-programming-guide.html#performance-tuning

[32] Apache Storm. [Online]. Available: https://storm.apache.org/

[33] Apache Storm Overview. [Online]. Available: https://sites.google.com/site/ownscratchpad/storm

[34] Michael Stonebraker, Uğur Çetintemel, Stan Zdonik. “The 8 Requirements of Real-Time Stream Processing”. [Online]. Available: https://cs.brown.edu/~ugur/8rulesSigRec.pdf

[35] Yahoo Benchmark. [Online]. Available: https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

[36] Yangjun Wang.” Stream Processing Systems Benchmark: StreamBench “, 2016. [Online]. Available: https://core.ac.uk/download/pdf/80719841.pdf

[37] A. Agarwal, M. Slee, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 2007.

[38] Exactly Once processing. [Online]. Available: https://www.ververica.com/blog/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink

[39] Splunk. [Online]. Available: https://www.splunk.com/en_us/blog/it/exactly-once-is-not-exactly-the-same.html

[40] MapReduce. [Online]. Available: https://en.wikipedia.org/wiki/MapReduce

[41] Metrics. [Online]. Available: https://metrics.dropwizard.io/4.1.2/manual/core.html

[42] Kryp Serialization. [Online]. Available: https://blog.knoldus.com/kryo-serialization-in-spark/

[43] Kafka at Linkedin. [Online]. Available: https://engineering.linkedin.com/blog/2019/apache-kafka-trillion-messages

[44] Kafka at Uber. [Online]. Available: https://www.confluent.io/resources/kafka-summit-2016/stream-processing-kafka-uber/

[45] Flink at Alibaba. [Online]. Available: https://www.alibabacloud.com/blog/why-did-alibaba-choose-apache-flink-anyway_595190

[46] Flink at Zalando. [Online]. Available: https://engineering.zalando.com/posts/2017/07/complex-event-generation-for-business-process-monitoring-using-apache-flink.html

[47] Spark Cloudsort Benchmark. [Online]. Available: https://databricks.com/company/newsroom/press-releases/databricks-sets-new-world-record-cloudsort-benchmark-using-apache-spark-1-44-per-terabyte

[48] Spark at Facebook. [Online]. Available: https://databricks.com/session/scaling-apache-spark-at-facebook

[49] Lambda Architecture in Big Data. [Online]. Available: https://www.cuelogic.com/blog/lambda-architecture-in-big-data

[50] Kappa at Uber. [Online]. Available: https://eng.uber.com/kappa-architecture-data-stream-processing/

I. License

Non-exclusive license to reproduce thesis and make thesis public

I, Mansur Alizada

1. 1. herewith grant the University of Tartu a free permit (non-exclusive license) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, Real time vs micro-batching in streaming data processing: performance and guidelines,

supervised by Pelle Jakovits, PhD.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons license CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive license does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Mansur Alizada

14/MAY/2021