FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Real-time Data Stream Processing System
Student:	Ing. Vitalij Kozlov
Supervisor:	Ing. Michal Valenta, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of winter semester 2021/22

Instructions

The recent evolution of sensor technologies, wireless communication and other real-time sources is accompanied by rapidly growing demand for real-time applications, which rely upon instantaneous input and fast analysis being translated into decision or action within a short, often specific timeline. The aim of the thesis is to design a system that tackles unique aspects of streaming data on one of the areas that have made streaming data interesting – social media streams.

1. Analyze current solutions, trends and tools in the real-time large-scale data processing.
2. Design stream processing system with a focus on high availability, low latency and horizontal scalability.
3. Create a cloud-based system prototype utilizing social media data streams and selected analysis techniques as a proof of concept. Implement a delivery mechanism in a form of a web-based interface.
4. Perform system testing and evaluate the results.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague June 25, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Real-time Data Stream Processing System

Ing. Vitalij Kozlov

Department of Software Engineering
Supervisor: Ing. Michal Valenta, Ph.D.

January 8, 2021

Acknowledgements

I want to thank my supervisor Ing. Michal Valenta, Ph.D. for encouraging me to work on this topic and to Nathan Marz for filling me with enthusiasm about the topic.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on January 8, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Kozlov, Vitalij. *Real-time Data Stream Processing System.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Poptávka po zpracování big data se v posledních letech neustále zvyšuje navzdory přetrvávajícím výzvám. Data různých objemů i formátů je nutné získávat, ukládat a zpracovávat. Systémy pro zpracování dat musí být uzpůsobeny požadavkům big data, je tedy potřeba, aby byly spolehlivé, škálovatelné a udržitelné. V poslední době náročnost požadavků ještě vzrostla, jelikož data musí být zpracovávána nejen ve velkých objemech, ale i v reálném čase. Tato diplomová práce navrhuje design systému pro zpracování big data, který se soustředí na vysokou dostupnost, nízkou latenci a horizontální škálovatelnost. Systém je inspirovaný zavedenou architekturou Lambda, ale odráží také technologické trendy, jako jsou cloud computing na veřejné platformě nebo kontejnerizované aplikace. Tato práce také popisuje implementaci prototypu navrženého cloudového systému a zhodnocuje dosažené výsledky.

Klíčová slova Big Data, proudové zpracování dat, Lambda architektura, Apache Spark, Apache Kafka, Kubernetes, analýza v realném čase

Abstract

Big data processing has been in high demand in recent years despite its many challenges. Data in different volumes and formats needs to be collected, stored

and processed. To address the nature of big data, data systems must be designed in a reliable, scalable and maintainable manner. Over the last few years, the requirements have become even more demanding, as the data needs to be processed not only on a large scale, but also in real time. This thesis proposes a system design focused on high availability, low latency and horizontal scalability. It is inspired by well-established Lambda architecture but also reflects certain technological trends, such as public cloud computing and container technology. This thesis also describes the implementation of a cloud-based system prototype that was created as part of the project and discusses its results.

Keywords Big Data, Stream Processing, Lambda Architecture, Apache Spark, Apache Kafka, Kubernetes, Real-time Analytics

Contents

List of Figures

List of Tables

Introduction

In the 1800s, the discovery of gold in California set off a frenzied Gold Rush. Three hundred years later, the madness is back. And so is mining. Although it is not gold what is mined today. It is data.

The last 15 years brough a data revolution. Enormous quantities of data are produced every single day. The volume of machine, transactional or social data is overwhelming. Big companies recognized the potential hidden in the mountains and streams of data that was pouring into storages from all possible kinds of sources. Yet it was a challenge to tame this data. It required huge infrastructure investments and teams of dedicated engineers to operate data systems in order to collect, process and store the data in such a quantity.

It did not take long for the situation to change. Today, thanks to the rise of public clouds, technological progress and battle-tested methodology, data systems at scale can be operated by small companies or even individuals in a cost-efficient manner.

Aim of the thesis

The aim of this thesis is to identify current trends in big data processing, understand their concepts and reason about their success. This knowledge will be applied to propose a design of a complex data system with focus on stream processing. The design will meet some of the key requirements for such a system: high availability, low latency and horizontal scalability. The next step will be to implement a prototype of the system as a proof of concept, deploy it on a public cloud, evaluate it and discuss its quality.

Thesis structure

In the first chapter of the thesis the core concepts connected to large scale data processing will be explored and presented. The chapter will help the reader to

understand the challenges connected to data processing, the difference between batch and stream processing and introduce some of the most significant tools for data processing evolution.

The second chapter will present the requirements connected to data system design. It will then discuss specifics of stream processing systems and the most important architectures for such a system.

The third chapter contains requirements set for the prototype to be successful. All the specifics of the design will be explained to the reader in the next part of the chapter, along with the decisions behind the design of every system component.

The fourth chapter will lay out the details of the prototype implementation.

The last chapter evaluates the prototype, discusses its performance details, assesses all the requirements and proposes potential future improvements.

Chapter 1

From Batch to Stream Processing

1.1 The age of data

1.1.1 What is data, and why is it important?

We live in the data age, where data means facts, insights, knowledge and power to control outcomes. We are surrounded by data. We always have been, but the advancements in informatics in the past decades allowed a variety of new data sources to be born. Besides affecting the amount of produced data, it introduced new methods how to efficiently store this data, process it and transform it into valuable insights.

The terms "data" and "information" are often used interchangeably, although their meaning is different. Data can be thought of as a unit of information, thus the real value of data is tightly coupled to the value of information that can be extracted from it. Information can improve decision-making, power scientific progress and economic prosperity, essentially improving people's lives.

1.1.2 Big Data

Big data is a relatively new term which has been gaining extreme popularity in the last couple of years. As pointed out by Kleppmann (2017) in [5]: "It is so overused and underdefined that it is not useful in a serious engineering discussion." The term is, indeed, quite abstract and doesn't come with the unified definition, specifying how big the data should be to be considered "Big". In this work, we will use the term for "datasets that are relatively large to be stored in a traditional database system or processed by traditional data-processing pipelines." as suggested by Yarabarla (2017) in [14]. The datasets that belong to this category can usually scale to terabytes or even petabytes

of structured, semi-structured or unstructured data. The term is also often described by three V's – volume, velocity and variety. Saxena and Gupta (2017) in [15] add two more V's to the definition - veracity and value. The authors describe 5V's as:

- **Volume**

  This dimension refers to the amount of data. In the past, storing terabytes of data would have been a problem, but new technologies have eased the burden.

- **Velocity**

  Velocity refers to the data generation and movement speed that must be dealt with in a timely manner. It is an important dimension for data coming from online systems, sensors or social media.

- **Variety**

  This dimension tackles the fact that the data can come in all sorts of formats ranging from structured data that can be stored in traditional databases to unstructured data such as images or video files.

- **Veracity**

  Veracity describes how correct and valid the data is. Examples of data with veracity include Facebook and Twitter posts with nonstandard acronyms or typos.

- **Value**

  As the name suggest, this dimension describes the value that the data holds. It presents the biggest motivator to store and process the data.

1.1.3 Modern data-processing requirements

The described nature of big data brings some challenges if the data is to be processed on full scale, within acceptable time frame and to provide consistent results.In order to handle big data. Some of the most important points that need to be addressed are:

- **Scalable infrastructure**

  Scalable data platforms are able to accommodate rapid changes in the growth of data, either in traffic or volume.

- **Complex processing**

  As the data comes from many sources in the different formats and volumes, modern data processing may consist of multiple layers and include complex data flows.

- **Faster processing**

  Working in real-time becomes fundamental in today's world. In some cases, data freshness may be more important than the amount or size of data. The value of data may decrease with every second, which brings the importance of fast processing capability.

- **Continuous data flow**

  "Business-critical applications should continue running without much impact even when there is a system failure or multiple system failures (server failure, network failure, and so on). The applications should be able to handle failures gracefully without any data loss or interruptions."[14] (Yarabala, 2017)

- **Visible, reproducible analysis**

  The importance of data science has risen in the recent years. As more and more companies utilize data-driven approach in decision making, it is crucial to ensure that consistent, reproducible results are produced.

1.1.4 Technological trends and their effects on Big Data systems

In order to approach challenges connected to Big Data, it is helpful to understand recent trends in technology, as they can significantly influence the way in which big data systems can be built.

First of all, we've started to hit the physical limits of how fast a single CPU can go. [1] That means that in order to scale data system efficiently, parallelization is to be heavily utilized. Instead of upgrading a single machine, known as *vertical scaling*, the systems are scaled by adding more machines. This is known as *horizontal scaling*.

Another trend we witness is the rise of public clouds. Public clouds offer renting of hardware in multiple locations on demand. This allows companies to operate their systems without the need to own costly hardware. Public clouds let their users optimize the number of machines in the cluster, enabling access to scalability and potential cost savings.

Besides the access to affordable hardware, a variety of software solutions was produced over the past few years by the open-source community. Some of these projects have matured and became industry standards, offering its users the option to operate performant, battle-tested software for affordable costs.

1.2 Distributed computing and large-scale data processing

1.2.1 MapReduce paradigm

MapReduce is a distributed computing paradigm that provides primitives for scalable and fault-tolerant batch computation.[1] Kleppmann (2017) in [5] describes the importance of MapReduce for Biga Data processing as "a major step forward in terms of the scale of processing that could be achieved on commodity hardware." Besides scalability, MapReduce offers the ability to parallelize the computation automatically, as it focues on *what* needs to be computed instead of *how* to compute it. In [1] Marz (2015) explains that: "All the details of concurrency, transferring data between machines, and execution planning are abstracted for you by the framework". As the paradigm name suggest, its backbone is made up by *map* and *reduce* functions that manipulate key/value pairs. These primitives are expressive enough to implement a large variety of functions.

The canonical MapReduce example is a *word count*. It is a program that takes a dataset of text and computes the number of times each word appears in it. As the input data to the program is stored within a distributed dataset, the first step is to determine which machines in the cluster host the blocks containing the data. After determining the input location, the data can be processed. MapReduce launches a number of map tasks proportional to the size of the input data. Every task is part of the input and executes map function on the assigned data. MapReduce promotes data locality. As stated by Marz (2015): "Because the amount of the code is typically far less than the amount of the data, MapReduce attempts to assign tasks to servers that host the data to be processed". The function generates intermediate key/value pairs as shown in Figure 1.1. In the word count example, the keys are words, which are assigned value of 1.

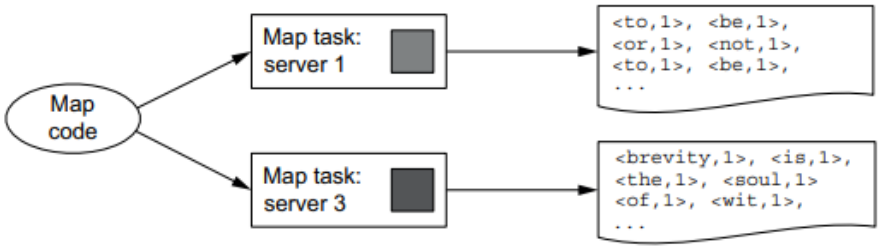

Figure 1.1: Map phase of MapReduce program [1]

Besides map tasks, there are also reduce tasks spread across the cluster.

These tasks are responsible for computing the reduce function for a subset of keys produced by the map function. Since the function requires all the values for a given key, the reduce phase can be started only after the map phase is finished and data is redistributed across the cluster in such a way, that all the intermediate key/value pairs are co-allocated with the reduce task responsible for its processing. The data transfer associated with fulfilment of this requirement is called a *shuffle*. After the data is shuffled, every reduce task sorts its assigned key/value pairs and performs reduce function. In case of word count example, the values are summed together, as demonstrated in Figure 1.2.

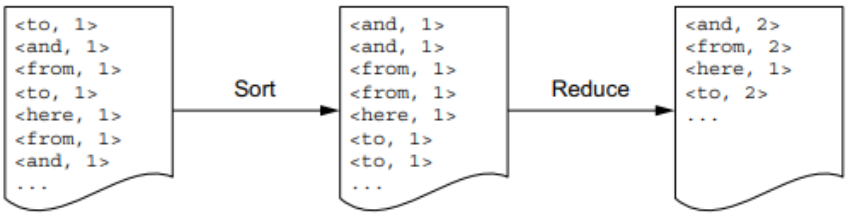

Figure 1.2: Reduce phase of MapReduce program [1]

The word count example demonstrates the key benefits of the model:

- Both map and reduce functions can be executed in parallel across the cluster.
- The program can be run in a distributed fashion.
- The complexity connected to concurrency and data transfers is handled for the programmer.

1.2.2 Hadoop

MapReduce model was widely popularized by Hadoop, as its approach showed to be efficient for data processing tasks. Apache Hadoop is a project focusing on creating general-purpose storage and analysis platform for big data. [2] Today, Hadoop is widely used in mainstream enterprises. The term *Hadoop* is often used by both core models of the Apache Hadoop framework and a variety of packages that can be installed on top of it. The core of the project includes five modules[16]:

- **Hadoop Common** The common utilities that support the other Hadoop modules.

- **Hadoop Distributed File System (HDFS)** A distributed file system that provides high throughput access to application data.
- **Hadoop YARN** A framework for job scheduling and cluster resource management.
- **Hadoop MapReduce** A system for parallel processing of large data sets.
- **Hadoop Ozone** An object store for Hadoop.

1.2.2.1 HDFS

White (2015) in [2] describes HDFS as "a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware". By "streaming data access patterns" White implies that the file system is built around the idea of "write-once, read-many-times" pattern. It doesn't require expensive, highly reliable hardware and can handle petabytes of data.[2]

HDFS offers scalability and enables parallel processing, as files are spread across multiple machines. The file system also provides fault tolerance, as file blocks are replicated across the nodes.

1.2.2.2 Hadoop YARN

Apache YARN (Yet Another Resource Negotiator) is Hadoop's cluster resource management system. YARN was introduced to improve the MapReduce implementation and provide APIs for requesting and working with cluster resources. [2] The provided APIs are typically not used by programmers directly, but are rather called by distributed computing frameworks, which themselves are built on YARN. The situation is illustrated in Figure 1.3. YARN brings in the concept of central resource management, which allows to efficiently run multiple applications on Hadoop.

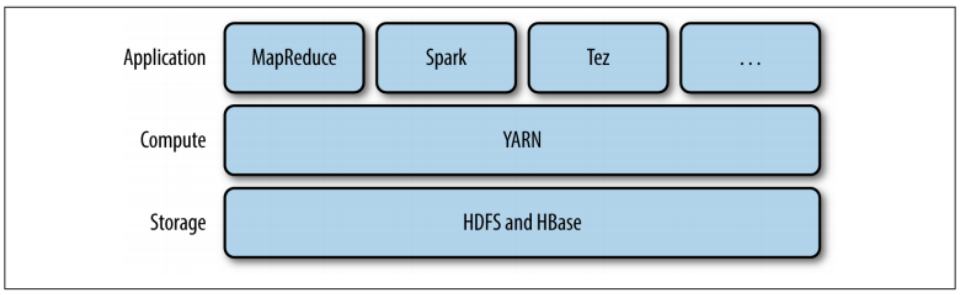

Figure 1.3: YARN application [2]

1.2.2.3 Hadoop shortcomings

Hadoop allowed to store and process the large volumes of data in a scalable, fault-tolerant manner. However, it has its own shortcomings. As HDFS is optimized for delivering high throughput, it will not work well if low-latency data access is required, as pointed by White (2015). Hadoop also introduces processing overhead since it involves a lot of read and write disk operations.

1.2.3 Apache Spark

Hadoop's shortcomings were partly addressed with the introduction of a new framework – Apache Spark. Spark was created for distributed general-purpose computing as a response to limitations in MapReduce cluster computing paradigm, which involved unnecessarily high number of disk reads and writes.

Spark is used by writing a *driver program* which implements high-level control flow of the application and launches various operations in parallel. Spark's programming model provides two main abstractions for parallel programming – resilient distributed datasets (RDDs) and parallel operations on these datasets. RDDs are read-only collections of objects that are distributed over a cluster of machines[17]. These objects are resilient, meaning they are fault-tolerant and can recover from failures by tracking the ancestry of data (framework keeps track of how a given piece of data is computed) [5]. As their name implies, RDDs are also distributed - their data resides in-memory across cluster nodes.

As stated in [17], RDDs allow several parallel operations that can be performed on them:

- ***reduce:*** Combines dataset elements using an associative function and produces result for a driver program.
- ***collect:*** Sends all elements of the dataset to the driver program.
- ***foreach:*** Passes each element of the dataset through a user provided function.

The described model allows it to have much better performance for algorithms that have to repeatedly iterate over the same dataset (as Spark is able to cache that data in memory rather than read it from disk every time. It's been shown that Spark can outperform Hadoop by 10x in iterative machine learning jobs. [17]

As already mentioned, the application logic of Spark program is implemented through a driver program, which acts as a controller of the application execution and maintains all of the state of the Spark cluster (the state of tasks and *executors*). The Spark executors are the processes that perform tasks assigned by the Spark driver. Their responsibility is to take the tasks assigned by the driver, run them, and report their state and results.[18]

Specifically, to run on a cluster, the driver program can connect to several types of *cluster managers*, which allocate resources across applications. This is demonstrated in Figure 1.4. Once the driver program is connected to the cluster, Spark acquires executors on cluster nodes through cluster manager. Next, the driver program sends the application code and tasks to the executors, which can start running them.

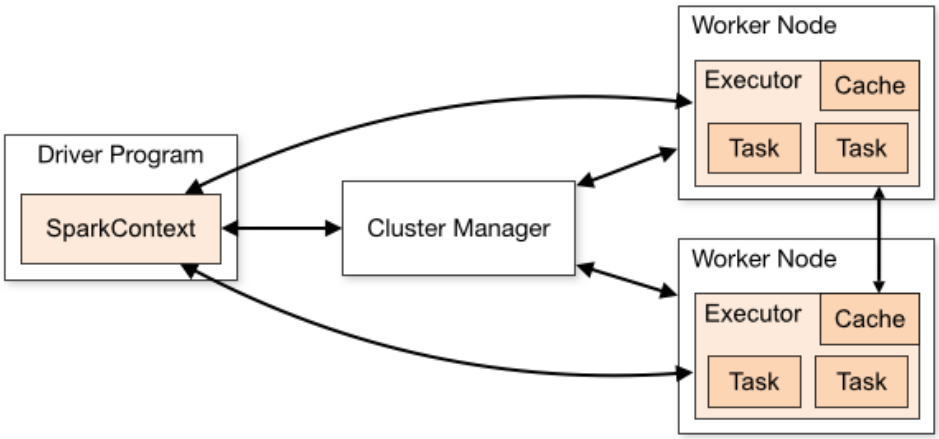

Figure 1.4: Spark application running in cluster mode. [3]

Spark currently supports several cluster managers[17]:

- **Standalone** – a simple cluster manager included with Spark.
- **Apache Mesos** - a general cluster manager that can also run Hadoop MapReduce and service applications.
- **Hadoop YARN** - the resource manager in Hadoop.
- **Kubernetes** - an open-source system for automating deployment, scaling, and management of containerized applications.[1]

Another option Spark offers is to specify where the driver and executor processes will be physically located by setting the *execution mode*. The three modes Spark supports are[18]:

- **Cluster mode**

  Cluster mode is probably the most common way of running Spark applications. In this mode, a user submits his code to a cluster manager,

[1]Kubernetes is the latest supported cluster manager. The support was introduced with Spark 2.3.0 in 2018 [19]

which launches the driver process on one of the worker nodes in the cluster, in addition to the executor processes. All the processes run in the cluster in this scenario.

- **Client mode**

  In client mode the driver process is run on the client machine that submitted the application. These machines are also known as *gateway machines* or *edge nodes* and are responsible for maintaining Spark driver process.

- **Local mode**

  Local mode allows to run Spark applications on a single machine, achieving parallelism using threads. It provides a way to experiment with Spark or test Spark applications.

1.2.4 Big Data and Kubernetes

It is not an accident that Kubernetes have become the latest supported resource manager by Apache Spark. The progress moves with unbelievable speed and new requirements arise every day. Although it's been only few years since Hadoop became mainstream, it already starts lagging behind in satisfying the current needs in Bag Data. Anadiotis (2018) [20] describes the situation as follows: "Hadoop was built in a world with different fundamental assumptions than the world we live in today. A world in which network latency was a major bottleneck, and cloud storage was not a competitive option."

Today, the situation is different. Network latency is less of an issue, which contributed to the rise of private clouds. Companies are tempted by scalable, cost-efficient services they are offered and are moving their operations either to public or hybrid clouds. Flexibility becomes an important aspect, where the ability to quickly deploy, scale or move the infrastructure is vital.

Another shift we can witness in the last few years concerns the whole application development process and how applications are taken care of in production. Luksa (2018) in [4] comments the situations as follows: "Organizations are realizing it's better to have the same team that develops the application also take part in deploying it and taking care of it over its whole lifetime. This means the developer, QA, and operations teams now need to collaborate throughout the whole process. This practice is called DevOps." Similar to how DevOps are changing the way software is developed, DataOps are changing the way data products are created with both methodologies focused on speed, quality and flexibility.

1.2.4.1 Container technology

One of the possible solutions to arising needs can be container technology. "Containers offer a logical packaging mechanism in which applications can be

abstracted from the environment in which they actually run. This decoupling allows container-based applications to be deployed easily and consistently, regardless of whether the target environment is a private data center, the public cloud, or even a developer's personal laptop." [21] The concept of containers allows developers to focus on the application logic without having to worry about the pitfalls of different environments, where it's run. It allows flexibility and can increase development speed.

In order to demonstrate container technology, it is usually compared to Virtual Machines (VMs). In case of VMs a guest operating system (OS) runs on top of a host operating system with virtualized access to the underlying hardware.[4] Unlike virtual machines, where processes run in separate operating systems, a containerized process runs inside the host's operating system. This makes containers far more lightweight: they share OS kernel, start much faster and use much less memory than VMs.[21] At the same time they offer the benefit of logical isolation – to the process itself, it looks like it's the only one running on the machine and the OS. The difference between isolation using containers and VMs is demonstrated in Figure 1.5.

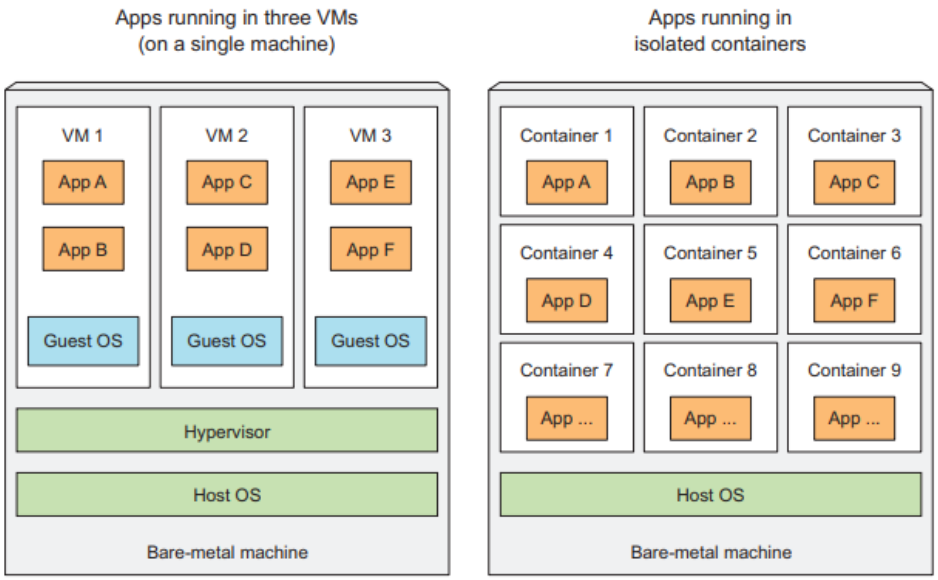

Figure 1.5: Isolating groups of applications using VMs vs. isolating applications with container [4]

1.2.4.2 Kubernetes core concepts

Thanks to their benefits, container technologies quickly became popular. However, with growing number of deployable application components in the sys-

tem, it becomes harder to manage them all. The need of cost-efficient, manageable deployment of components on a large scale gave birth to Kubernetes. Luksa (2018) describes Kubernetes as "A software system that allows you to easily deploy and manage containerized applications on top of it. It relies on the features of Linux containers to run heterogeneous applications without having to know any internal details of these applications and without having to manually deploy these applications on each host." It allows automating deployment, scaling, and management of containerized applications. Kubernetes does so by abstracting away the underlying infrastructure. Deploying applications through Kubernetes is always the same, whether it runs on a large cluster or a single virtual machine.

Figure 1.6 shows a simple view of Kubernetes system. The system consists of a master node and any number of worker nodes. The developer submits a list of applications to the Kubernetes master, which deploys them to the cluster of worker nodes. Kubernetes allows the developer to specify which applications have to be run together in order to be deployed on the same node by Kubernetes. Otherwise, it shouldn't matter on what node a component lands.

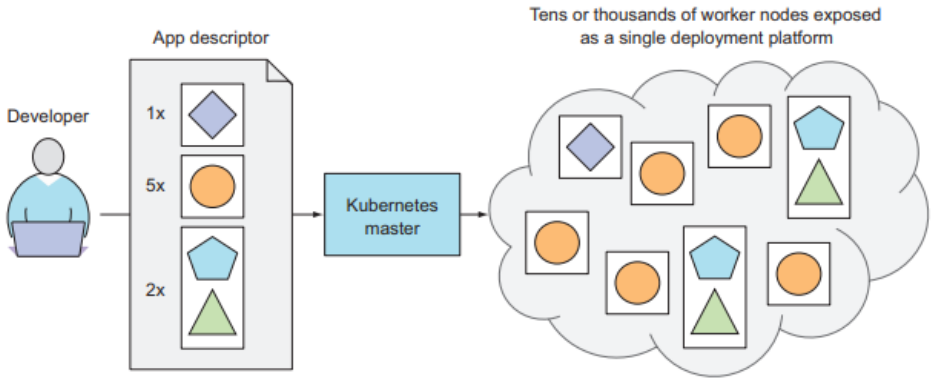

Figure 1.6: Kubernetes deployment illustration [4]

A closer look to the architecture of a Kubernetes cluster is shown in Figure 1.7. It shows two types of nodes in Kubernetes cluster - the master node, which holds the *Kubernetes Control Plane* that controls and manages the whole Kubernetes system and worker nodes thar run the actual deployed applications [4]. The Control Plane consists of multiple components which can be run on a single master node or be split across multiple nodes and replicated to achieve high availability. The components of the Control Plane include [4]:

- **Kubernetes API Server** – provides means to query and manipulate the state of objects in Kubernetes.

- **Scheduler** - is responsible for assigning worker nodes to each deployable component of the application.
- **Controller Manager** – performs cluster-level functions. These include component replication, keeping track of worker nodes or handling failures.
- **etcd** – a distributed data store which persists cluster configuration

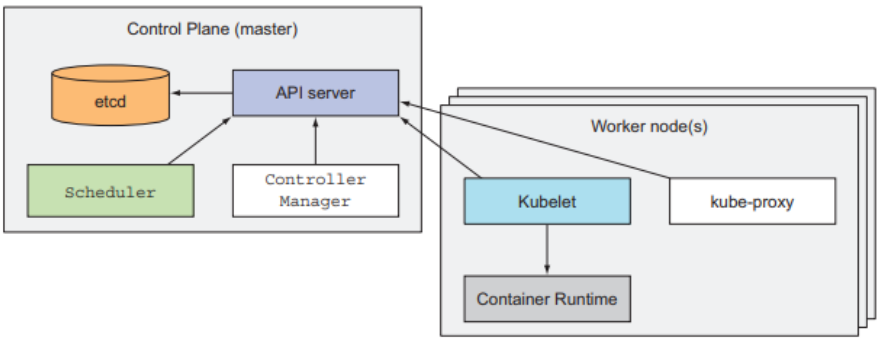

Figure 1.7: Kubernetes cluster components [4]

1.3 Towards Real-Time and Streaming Big Data

"The term "streaming" is used to describe continuous, never-ending data streams with no beginning or end, that provide a constant feed of data that can be utilized/acted upon without needing to be downloaded first." [22] Kelpman (2017) in [5] defines streams as: "data that is incrementally made available over time." We live surrounded by streams coming in various forms, volumes and from different sources. Data streams can be generated by network devices, server log files or social media users. The challenges connected to data streams include the ability to process, store, and analyze streams as it's generated real-time, which makes systems focused on stream processing highly demanding. Kleppmann (2017) in this context distinguishes three types of systems:

- **Services (online systems)**

  This type of service work in a reactive manner. A service waits for a request or instruction from a client to arrive. When one is received, the service tries to handle it as quickly as possible and sends response back to the requester. Response times and availability of the system are considered the primary measure of performance.

- **Batch processing systems (offline systems)**

  Batch processing systems take a large amount of input data, run a *job* to process it, and produce output data. Jobs typically take long time to complete (sometimes even days). Batch jobs are often scheduled to run periodically. The performance of these jobs is usually *throughput* (the time it takes to process the input of a certain size).

- **Stream processing systems (near-real-time systems)**

  Stream processing lies somewhere between online and batch processing. Stream processor consumes inputs and produces outputs (rather than responding to requests), just like batch processor. However, unlike batch job, which operates on a fixed set of input data, a stream job operates on events shortly after they happen. Stream processing systems are measured both by their latency and throughput.

1.3.1 Event streams

In stream processing context, a record is more commonly known as an *event*, but it is essentially the same thing. Kleppmann (2017) defines it as "a small, self-contained, immutable object containing the details of something that happened at some point in time." Events usually contain timestamp indicating when it occurred. An event might describe a sensor measurement, pageview or a creation of a new post on social media.

In batch processing, a file is written once and then potentially read multiple times. Analogously, in stream processing, an event is generated once by a *producer* (also known as *publisher* or *sender*), and then potentially processed by multiple *consumers* (*subscribers* or *recipients*)[5] A set of related records is often referred as a *topic*.

1.3.1.1 Messaging systems

In principle, a file or a database can be used to connect producers and consumers. However, it is not very efficient, since they lack notification mechanism and polling for updates can quickly become expensive.

One of the ways of notifying consumers about new events is using *messaging system*: a producer sends a message containing the event, which is then published to consumers. Kleppmann (2017) compares two types of messaging systems:

- **Direct messages**

  These systems use direct network communication between producers and consumers without intermediary nodes. The disadvantage of the approach is that they are prone to message loses. For example, if a consumer is offline, it may miss incoming messages.

- **Message brokers**

  Kleppmann (2017) describes broker as "database that is optimized for handling message streams." Brokers run server, with producers and consumers connecting to it. Producers write messages to the broker, and consumers read them from the broker. The approach tolerates clients that connect and disconnect, as responsibility of durability is moved to the broker.

One disadvantage of messaging systems is when a new consumer is added, it typically only starts receiving messages sent after the time it was registered. Any messages sent before the connection are already gone and can't be accessed.

1.3.1.2 Partitioned logs

Log-based message brokers are hybrids, combining durable storage with a low-latency notification system of messaging. In this approach producers send messages by appending it to the end of the log and consumers receive messages by reading the log sequentially. When a consumer reaches the end of the log, it waits for a notification that a new message has been appended [5].

Log-based message brokers allow scalability to higher throughput than a single disk can offer by utilizing *partitioned* logs. This approach is illustrated in Figure 1.8, where each partition can be read and written independently from other partitions, as they are stored on different machines. Topics are defined as group of partitions that carry messages of the same type.

Within each partition, the broker assigns monotonically increasing number called *offset* to every message[5]. As partitions are append-only, this approach allows to maintain them ordered. Consuming a partition sequentially allows to tell which messages have been already processed and which have not: all messages with an offset less than a consumer's current offset have been processed an all messages with a greater offset haven't been seen yet.

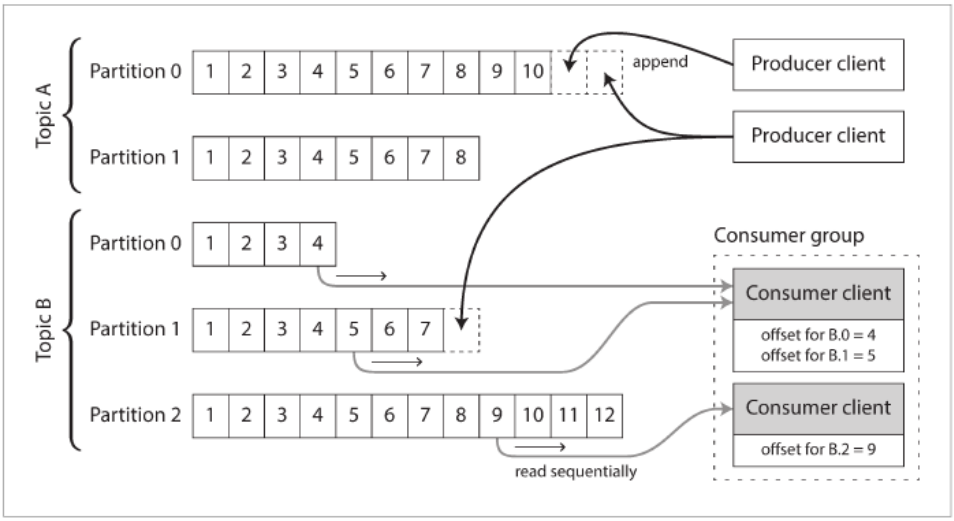

Figure 1.8: Producers send messages by appending them to a topic-partition file, and consumers read these files sequentially. [5]

1.3.2 Stream processing

Stream processing can be defined as "the act of continuously incorporating new data to compute a result." [23]. In stream processing the input data is *unbounded*. Unbounded data is "a type of dataset that is infinite in size (at least theoretically)" as stated by Maas (2019). The input data simply has no predetermined beginning or end. In contrast, by *bounded* data we understand a dataset of a known size.

1.3.2.1 Uses of stream processing

Stream processing is a key requirement in many big data applications. Some Stream processing use cases include real-time reporting, fraud detection or trading systems. Kleppmann (2017) mentions some of the important applications of stream processing:

- ***Complex event processing (CEP):*** an approach for analyzing event streams, especially geared toward searching for certain event patterns. Similarly to the way regular expressions allow to search for patterns in a string, CEP allows to specify rules to search for patterns in streams.
- ***Stream analytics:*** Stream analytics, unlike CEP, tends to be less interested in finding specific events, but is rather oriented towards aggregations and statistical metrics. This might include measuring event occurrences or calculating rolling average of a value per time interval.

- ***Search on streams:*** While CEP allows searching for patterns in multiple events, there is also a need to search for individual events based on complex criteria, such as full-text queries.

1.3.2.2 Execution modes

The two different techniques to process the streams are used today. Each of them has its own advantages and handicaps.

The first execution mode is known as *continuous processing*. In continuous-based systems, each node is continuously listening to messages from other nodes and outputting new updates to its child nodes. Chambers (2018) in [18] explains the idea on the implementation of map-reduce computation: "In a continuous processing system, each of the nodes implementing map would read records one by one from an input source, compute its function on them, and send them to the appropriate reducer. The reducer would then update its state whenever it gets a new record." Chambers (2018) also implies that continuous processing offers lowest possible latencies when total inputs are low, as each node responds immediately to a new message. However, continuous systems generally have lower maximum throughput, because they incur a significant amount of overhead per-record.

An alternative to continuous processing is provided by *micro-batch processing*. Micro-batch systems wait to accumulate small batches of input data (say, 500 ms' worth), then process each batch in parallel using distributed collection of tasks, similar to the execution of batch job. Chambers (2018) states in [18]: "Micro-batch systems can often achieve high throughput per node because they leverage the same optimizations as batch systems (e.g., vectorized processing), and do not incur any extra per-record overhead." The downside of this approach is, however, higher latency due to waiting to accumulate a micro-batch.

1.3.2.3 The effect of time

Stream processors often need to deal with time, especially when used for analytics purposes, which frequently use time windows such as counts and averages over a time period. Chambers (2018) in [18] states that "in stream-processing systems there are effectively two relevant times for each event: the time at which it actually occurred (event time), and the time that it was processed or reached the stream-processing system (processing time)." These terms can be described as follows:

> *Event time:* Time embedded in the data itself. In most cases it is not required to be precise time when the event actually occurred. The main importance of the event time is that it allows more robust way to compare events against each other.

Processing time: Time at which the stream-processing system actually receives data.

Many stream processing frameworks use the local clock on the processing machine (the *processing time*) to determine windowing. [5] This practice has the advantage of being simple, and it is reasonable if the delay between event creation and event processing is short enough. However, this approach may be problematic if some significant processing lag occurs. This fact is demonstrated in Figure 1.9 where measurement results are distorted due to processing lag caused by a restart.

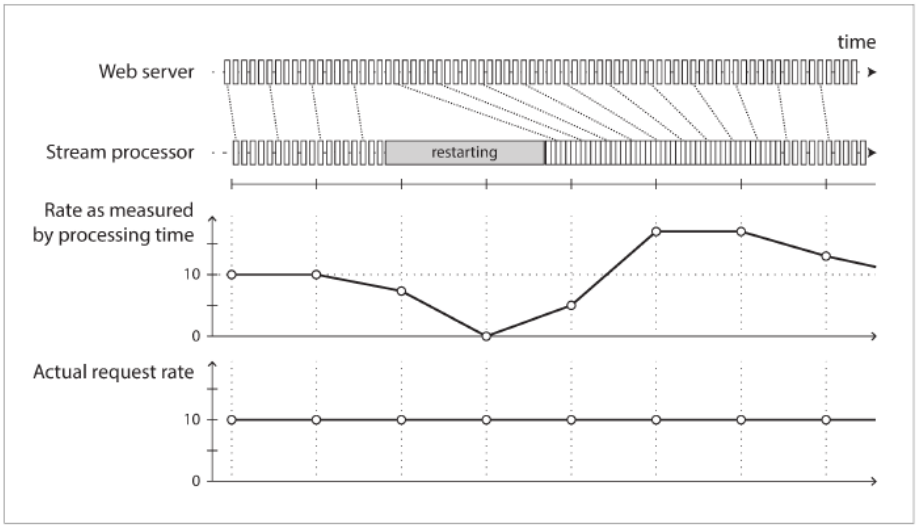

Figure 1.9: Inaccuracies in measurement introduced by windowing by processing time. [5]

Time window aggregations, such as the one demonstrated in 1.9 are widely used in stream processing. Kleppmann (2008) mentions several types of windows that are commonly used:

- ***Tumbling window:*** A type of window with a fixed length. Every event belongs exactly to one window.

- ***Hopping window:*** A hopping window has also a fixed length, however it allows windows to overlap in order to provide some smoothing. For example, a 5-minute window with a hop size of 1 minute would contain the events between 10:03:00 and 10:07:59, then the next window would cover events between 10:04:00 and 10:08:59, and so on.

- ***Sliding window:*** A sliding window contains all the events that occur within some specified interval of each other. For example, a 5-minute sliding would cover events 10:03:39 and 10:08:12 because they are 5 minutes apart. Note the difference opposing to tumbling and hopping windows, which wouldn't have put these two events in the same window because they use fixed boundaries.
- ***Session window:*** This type of a window, unlike others, has no fixed duration. It is defined by grouping together all events for the same user that occur closely together in time. The window ends when no user events occur for some period of time (for example when user is inactive for 30 minutes).

Chapter 2

Real-Time Data Processing Architecture

In the earlier days, big data systems were primarily constructed to handle 3 V's of big data: Volume, Velocity and Variety. Kleppmann (2017) in [5] lists many factors that may influence the design of a data system, including the skills and experience of people involved, time scale for delivery, legacy system dependencies, organization's tolerance for different kinds of risks or even regulatory constraints. But regardless the circumstances, there are three important aspects that the big data system needs focus on, according to the author:

- **Reliability**

  The system should continue to work correctly (performing the correct function at the desired level of performance) even in the face of adversity (hardware, software, and human errors).

- **Scalability**

  As the system grows (whether it is, data volume, traffic volume, or its complexity), there should be ways of dealing with that growth.

- **Maintainability**

  Many different people may work on the system over time (engineering, operations), maintain it, or adapt to new use cases. They should all be able to work on it productively.

2.1 Big data architecture components

Taming big data has always presented a challenge due to its nature. Efficiently collecting, storing and processing large amounts of heterogenic data required

a centralized approach, which would avoid all the pitfalls the data presents inside all its stages in the system. "Big data architecture refers to the logical and physical structure that dictates how high volumes of data are ingested, processed, stored, managed, and accessed."[24]. Kalipe et al. (2019) claim that: "A Big data architecture describes the blueprint of a system handling massive volume of data during its storage, processing, analysis and visualization."

Besides the requirements proposed by Kleppmann (2017), Ellis (2014) suggests that stream-processing systems have to share three key features: high availability, low latency and horizontal scalability. If batch-oriented systems become unavailable for minutes or even hours, it is unlikely to affect operations. Real-time systems, on the other hand, are sensitive to these sort of outages and may even be sensitive to scheduled maintenance windows. Besides that, the system needs to process data in a scalable manner and minimize time between the event enters the system and the moment it is available for delivery. These needs add another level of complexity when designing such a system. It is important keep all of these requirements in mind when selecting individual system components.

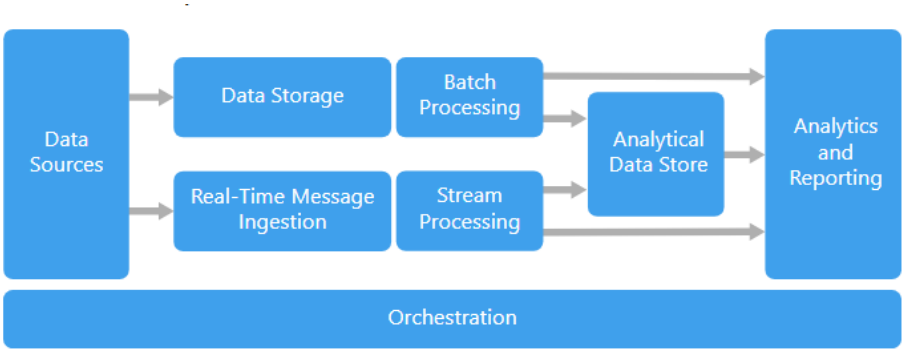

Figure 2.1: The Components of big data architecture [6]

When talking about Big Data architectures, most of them include some or all of the components listed by [6]. To illustrate their role in a data pipeline, see Figure 2.1. These components include:

- *Data sources:* Data sources include either static data, such are files produced by applications (e.g. logs), or real-time data sources, for example sensor data, financial trading data, or social media streams.
- *Data storage:* Their purpose is to store data for batch processing. The storages are usually built around the idea of "write-once, read-many-times". The amount of data in them typically grows with time, which requires data storage to be scalable and reliable.

- *Batch processing:* Batch processing allows to process data at a large scale, applying various transformations and aggregations and provide the output data for further analyses.

- *Real-time message ingestion:* Systems which focus on stream-processing must typically provide a way to capture and store these messages before they are processed. Message ingestion mechanisms provide a way to promote load-balancing and fault-tolerance.

- *Stream processing:* After real-time messages are captured, they are read and processed by filtering, aggregating and other methods preparing the data for analysis. The processed data is written to an output sink.

- *Analytical data store:* Analytical data stores provide the option to query the data computed by either batch or stream processing and retrieve it in a structured format.

- *Analysis and reporting:* Some system ease providing data insights through analysis and reporting. The architecture might include data modeling layer, presenting users interactive data explorations, drill-downs, etc.

- *Orchestration:* As most of the data processing and operations consist of repeating set of steps, they are usually encapsulated into workflows. This might include workflow management and deployment platforms that help scheduling processing tasks, manage workflow dependencies or event infrastructure deployments and upgrades.

2.2 Lambda architecture

One of the major breakthroughs in the real-time data-processing architectures was caused by Nathan Marz and his widely discussed article named "How to beat the CAP theorem" in 2011 [8]. In the article Marz discusses the complexity brought to data systems with the rise of distributed databases in combination with incremental algorithms (algorithms relying on updates in these distributed databases). The author proposes solution which relies on immutable data and a three-layered architecture that can handle massive quantities of data by taking advantage by both batch and stream-processing methods. The details of the proposed architecture appeared in the author's book, published in 2015 [1]. A representation of the architecture can be seen in Figure 2.2

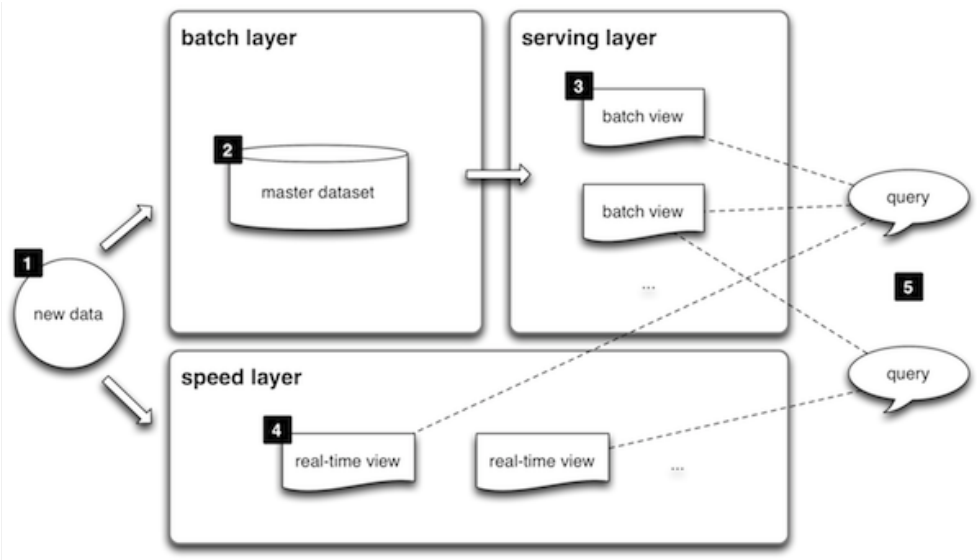

Figure 2.2: Lambda architecture [7]

2.2.0.1 Problems with fully incremental architectures

In [1] Marz (2015) points out the problems with architectures that use read-/write databases and maintain the state of those databases incrementally as new data arrives. Author provides an example of an incremental approach by counting pageviews. The approach is based on maintaining a counter and increment it every time a new pageview occurs. According to Marz the major disadvantages of the approach are:

- **Operational complexity**

  In a read/write database, as a disk index is incrementally added to and modified, parts of the index become unused. These unused parts take up space, which at some point in time needs to be reclaimed.

  Reclaiming space as soon as it becomes unused brings a lot of overhead, so it is reclaimed in bulks. A process of reclaiming the space is called *compaction*. Compactions are expensive operations, which demand a lot of machine resources, and may jeopardize machines' smooth functioning. Compactions need to be managed correctly by scheduling it on each system node in a way that not too many nodes are affected at once.

- **Complexity of achieving eventual consistency** Marz (2015) points out that: "A theorem called the CAP theorem has shown that it's impossible to achieve both high availability and consistency in the same system

in the presence of network partitions." To achieve high availability, distributed databases keep multiple replicas of the stored information. By doing so, the information is still available, when one of the database nodes goes down. The states of the replicas, however, may diverge during network partitions due to different sets of updates they receive. Only when the network partition is gone, the replicas can be merged. This puts a requirement on a data structure, if it is to be merged correctly. In general, handling eventual consistency in incremental, highly available systems is unintuitive and error-prone.

- **Lack of human-fault tolerance** An incremental system is constantly modifying the state it keeps in the database, which means a mistake can also modify the state.

2.2.0.2 Batch layer

The problem with the incremental approach is tackled in the *batch layer* of their architecture. Marz (2015) suggest using data model which relies on immutable data. The author introduces the *fact-based* model for representing data. By ensuring data immutability (not allowing updates or deletes) this model brings simplicity and human-fault tolerance. If we get back to the example with page-views, instead of updating the current state of the counter, a new record is created, representing pageview event as a fact with included timestamp. A collection of these fact collected over time is called *master dataset*.

The responsibility of the batch layer is to do two things: store an immutable, constantly growing master dataset and regularly compute arbitrary functions on that dataset. To avoid pitfalls of incremental algorithms, the views produced by the batch layer are not updated, but rather recreated from scratch. This type of processing is best done using batch-processing systems. It allows to process large volumes of data and apply complex functions on the data. This repeating process is showed in Figure 2.3.

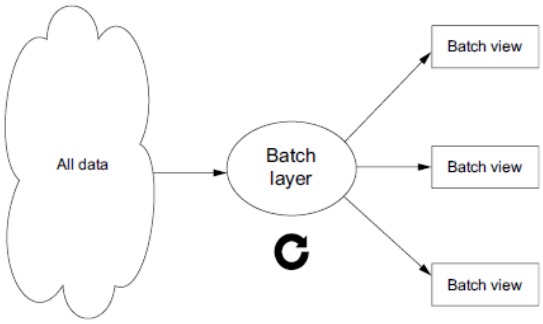

Figure 2.3: Lambda architecture [7]

2.2.0.3 Serving layer

Serving layer is the simplest one in the proposed architecture. As batch layer emits views as the result of its functions, the next step is to load the views somewhere, where they can be queried. The serving layer is represented by a specialized distributed database that loads batch views and allows to do random reads on them. Marz (2015)

2.2.0.4 Speed layer

So far, the system is able to efficiently store, process and serve the data. But the views computed by the layer provide only the data that was present in the master dataset before the start of the last batch run. To tackle this problem, a *speed layer* is proposed in the architecture.

To compensate for the delay caused by a long-running nature of the batch layer, another system is run in parallel with it and computes functions on the data in real-time. The goal of the layer, as its name suggest, according to Marz (2015) is: "to ensure new data is represented in query functions as quiclky as possible".

One big difference between batch and speed layers is that speed layer does incremental computation instead of recomputations in order to achieve the smallest latencies as possible.

Speed layer produces speed views which contain all the data needed to compensate for the batch views. Resolving query function requires merging batch and speed views, as illustrated in Figure 2.4.

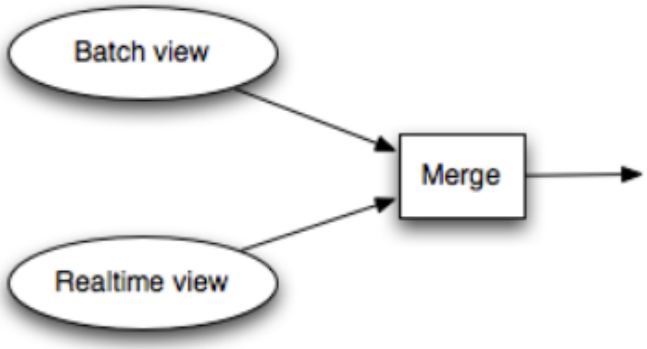

Figure 2.4: Computing a query in Lambda architecture [8]

2.3 Kappa architecture

Lambda architecture has quickly become popular, but certain criticism also emerged after some time. Jay Kreps (2014) in [9] acknowledges the benefits of Lambda architecture, but also implied it brings some drawbacks.

The first thing Kreps (2014) pointed out to, is the necessity of "maintaining two code bases that need to produce the same result in two complex distributed systems".

Another thing Kreps (2014) finds as a drawback is operational burden of running and debugging two systems.

As an alternative to Lambda architecture Kreps (2014) proposed using a simplified version of the architecture which does not include batch layer at all. The architecture overview that started to be called "Kappa" is demonstrated in Figure: 2.5. The approach is simple:

1. Using a system that helps to retain full log of the data that has to be potentially reprocessed and which allows for multiple subscribers. As an example of such a system Kraps names Apache Kafka[2]

2. When reprocessing needs to be done, a second instance of stream processing is started. This jobs starts processing data from the beginning of the retained data and directs the output to a new output table.

3. When the second job has caught up, the application is switched to read from the new table.

4. The old version of the job is stopped and its output table is deleted.

[2]Apache Kafka is a log-based message broker that utilizes partitioned logs.[5]

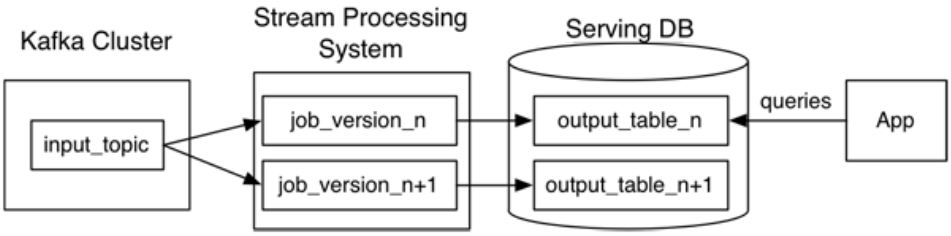

Figure 2.5: Kappa architecture [9]

Chapter 3

System Analysis and Design

3.1 System specification and requirements

Building reliable, scalable and maintainable data-intensive systems can be challenging even for experienced teams of engineers. In this work a prototype of such a system is proposed as a proof of concept, although production-ready quality is not expected. In this section some prototype requirements will be set. In later chapters we'll evaluate how well these requirements were met.

The system should focus on collecting, processing and analyzing data from microblogs and unlocking new insights from it. Analyzing data from social media brings many challenges and unpredicitable future directions. One of the main features of social data is its dynamics, making latest data very valuable or minutes old data worthless. This is why near real-time processing of this kind of data is required. With stream processing it becomes desirable not only to process the newest data, but also to be able to act upon it, apply predictive modeling, etc. This usually requires extracting information from real-time data and combining it with historical data for a given context on the fly.

With this information, it is clear that the system should provide fast access to both real-time and historical data. It should be flexible, allowing to incorporate new data sources, changing schemas and data formats. The system should be able to project potential changes in business requirement and processing methods affecting output data on both real-time and historical results. If, for instance, a new, improved version of machine learning model used in the data processing pipeline is utilized, it should affect not only the outputs based on data processed after the transition to a new model, but all the historical results as well. Increased loads should be handled without compromising system performance. Hardware failure effects on the system operation should be minimized.

3.1.1 Functional requirements

FR1 - Efficient data storage

The system offers flexibility to decide how to store and compress large volumes of data to suit specific needs.

FR2 - Fast access to historical data for real-time context

The system is able to efficiently enrich real-time data with the relevant data from the past based on predefined logic.

FR3 - Datasets are queryable at any time in their history

The selected data model allows to query the data for any time range covered in the dataset and reproduce query results as if it was run at any given timestamp in history.

FR4 - Ad hoc queries support

The system provides easy access to the data and the ability to arbitrarily mine it on a large scale in a cost-efficient way.

FR5 - Addable data sources and formats

The system allows to incorporate additional sources of input data without impacting existing data and processing pipelines.

FR6 - Building new views

The data model allows computing and maintaining multiple views based on existing and newly formulated requests. Tracking of metrics covered by dataset can be done on-the-fly.

FR7 - Adaptive views

It's possible to introduce view updates, projecting new business logic, newly trained machine learning model results or other changes in the view computation. The new computational logic can be applied both on newly arrived and historical data.

FR8 - Delivery mechanism

The system provides means to continuously deliver its output data to the end user.

3.1.2 Non-functional requirements

NFR1 - Scalability

The system is able to maintain performance when facing increased load by adding system resources.

NFR2 - Low-latency processing

The input data is time-efficiently processed and is ready for delivery with a minimal delay.

NFR3 – Fault tolerance

The system provides means to face both hardware failures, and errors caused by human mistakes.

NFR4 - Generalization

Wide range of applications can be supported by the system. The system remains efficient even in case of changing requirements for data sources and characteristics or its output metrics.

NFR5 - Extensibility

Future system changes can be made. The system is adaptable to unanticipated use cases or evolving requirements.

NFR6 - Cloud platform compatible

The system can be deployed and operated on cloud-based platofrms.

3.2 Sample application

In order to showcase proposed system's features and benefits, a sample application utilizing the system will be built. The application represents a simple example of adoption of stream processing on a real data coming from one of the largest microblogs – Twitter. Its main purpose is to demonstrate functionality of the system and to help reader's understanding of problematics by showcasing it on real-life examples.

The application will leverage a system design described in the next section to collect, process and deliver potentially useful information contained in short posts (also known as tweets) created by users of Twitter social networking service. Twitter has become a major platform for spreading news, sharing thoughts and socializing with other people globally. With 353 million MAU (monthly active users) in Q3 2020, which produce over 500 million tweets a day [25], Twitter presents a data source with high potential of being utilized

for analytical purposes. Interpreting such a high-volume, high-velocity data stream may be beneficial across multiple domains. One of the topics which has caught researchers' interest is trend detection and monitoring based on tweet data. Natural language processing techniques allow to extract entities such as people or organizations from tweets and calculate the sentiment expressed by these tweets, unveiling details about both popularity and reputation of entities. This information might become helpful in case of predicting stock prices or even presidential election results. [26]

One of the Twitter's features called "hashtag" allows users to tag parts of tweet text, which enables cross-referencing of content. By using a hashtag symbol (#) before a keyword or phrase people categorize their tweets, allowing other users to find it more easily. A hashtagged word also becomes a link to a feed of tweets labeled by the hashtag, allowing users to see other tweets containing the same hashtag. Hashtagged words that become very popular can be found in a Twitter section called "Trends", where emerging topics of discussion appear to be found by Twitter users. An example of trending topics can be seen in Figure 3.1.

Figure 3.1: Example of Twitter Trends [10]

Hashtags will be the main subject of the sample application. The first challenge is low latency hashtag retrieval from tweets in a scalable, fault-tolerant manner. A similar functionality to Twitter's Trends will then be implemented - a list of trending hashtags will be produced and continuously updated from incoming stream of collected tweets, allowing application users to see hashtag rankings based on a number of occurrences in tweets during a given amount of time (e.g., last 10 minutes) with a minimal delay. On top

of that, the application will tackle the need of fast access to historical data for a given real-time context (trending hashtags) and will be able to retrieve time-series data representing tweet occurrence evolution for any hashtag on the fly. This data can be beneficial not only for the application user, but also for burst detection algorithms, which can distinguish topics that have been popular for a while or on a daily basis from the ones that become popular unexpectedly (natural catastrophes, sports events, etc.). Both trends and hashtag time-series charts will be accessible through a web-based dashboard to demonstrate "live data" delivery utilizing server push and fast time-series data retrieval.

3.3 System design

After analyzing system requirements, it is clear that many problems connected to their successful fulfillment are not new to the systems dealing with large volumes of data and stream processing. Some of these problems are tackled by the architectures described in the previous chapter. On the other hand, some of the architectures may not be suitable for our particular needs or may be unnecessarily complex. It is, therefore, essential to think about architecture strengths and weaknesses considering the particular use case.

First of all, the need for the system to be fault-tolerant, scalable and provide processed data with a low latency is a general requirement for a streaming system and is promised by all of the aforementioned architectures. Secondly, the system should provide not only real-time, but also historical data, which eliminates architectures that focus solely on stream processing without data retention. On top of that, it should be adaptive and support frequent requirement changes, which leads us to Nathan Merz's Lambda architecture. It will be the main inspiration for the system design, although some changes will be proposed. One of the main challenges will be to transfer the architecture, which was tailored for Hadoop ecosystem, to a public cloud platform and try to utilize its benefits, while eliminating Hadoop drawbacks.

In the world of big data, Apache Hadoop has been the reigning framework for deployment of scalable and distributed applications.[11] The rise of cloud computing, however, diminished Hadoop popularity, as can be seen from Figure 3.2, which compares search interest between Hadoop and Kubernetes on Google. It should be noted, that this comparison is not entirely fair, as Kubernetes doesn't present Hadoop direct substitute – in fact, Hadoop can be deployed on Kubernetes. Hadoop basically provides three main functionalities: a resource manager (YARN), programming paradigm (MapReduce) and data storage layer (HDFS). All three of these components are being replaced by more modern technologies, with one of them being Kubernetes for resource management.

MapReduce is being replaced by Spark/Flink and other processing tools, while cloud object and file storage services slowly replace HDFS.[11].

Figure 3.2: Comparison of Apache Hadoop and Kubernetes search interest [11]

In the proposed design we want to take advantage of the mentioned emerging technologies and create a system prototype that utilizes the benefits of public cloud computing. Besides no acquisition costs (no need to purchase hardware) and lower operating costs (you pay only for the resources and services you use), public clouds allow near-unlimited resource scalability and ease of a cloud maintenance. On top of that, running Kubernetes on such a cloud allows applications to scale resources up and down with a simple command, auto-scale based on usage and make the most economical use of computing, networking and storage resources.[27] A key benefit for operation teams is infrastructure abstraction – it can be configured once and run everywhere, easily shut down and redeployed when needed again. Thanks to public clouds and Kubernetes, big data systems become affordable for small companies or even individuals for personal usage.

In the rest of this chapter individual system components will be discussed in detail. We will take a closer look at the architecture layers, identify their connections to the system requirements and propose a toolset for building the system.

3.3.1 Message queueing

To understand the need of persistent queuing in the system, we will first consider an architecture without it. In such a system, messages would be handed directly to workers to process each message independently. But what would happen if a worker dies before completing its task? There is no inherent mechanism do detect or correct the error. The architecture is also prone to failures during traffic bursts that exceed the resources of the processing cluster. Writing messages to a persistent queue addresses these issues, as queues allow the system to retry message processing in case a worker fails and provide a place for messages to buffer when downstream workers reach their

processing limits.[1] Application of queueing also ensures that communication between message producers and message consumers becomes asynchronous. A producer doesn't need to wait until the message is processed, but waits only for a confirmation from a message broker that the message has been buffered. The processing of the message will happen at some undetermined future point in time. With the velocity at which social media data is produced and traffic bursts that can be expected with this kind of data, using persistent queueing would help to address the requirement for the system to be fault-tolerant. In order to achieve queue scalability and higher throughput than a single disk can offer, we want to use a partitioned log for the message storage. Different log partitions can be hosted on different machines, making each partition a separate log that can be read and written independently from the other partitions.[5]. Using logs is also a good way to ensure the workload is distributed across multiple consumers sharing the work. Load balancing is achieved by assigning entire log partitions to nodes in consumer group by the broker, instead of assigning individual messages.

Apache Kafka is a log-based message broker, which is scalable, is able to segregate load in multiservice ecosystems and ensures messages are durable. It provides load-balancing, allows to achieve strong ordering guarantees and is secure.[28] In the proposed system Apache Kafka will be used to help the system handle unexpected loads of social media messages without jeopardizing workers by exceeding their processing resources. It will act as a temporary storage for these messages to allow workers to continue processing after unexpected crashes right where they left before the crash. Kafka will also help feeding consumers sharing the work in parallel by utilizing the partitioned logs.

3.3.2 Batch layer

One of the main reasons the system design was inspired by the Lambda architecture, is its ability to handle large volumes of data by taking advantage of both batch and stream processing methods. Our system requirements state that the system should provide access to both real-time and historical data. This means that the system should provide the functionality of persistent data storage and the ability to query the data. On top of that, the system should be able to reproduce query results as if the query was ran in a given point in time, which leads us to the necessity to select a right model for data storage. The fulfillment of this requirement will allow us to reprocess the data at any time, enabling introduction of new metrics or adjusting the definition of the existing ones as a reaction to changing business logic or processing methodology. As a result, the system will not only reflect methodology changes on the newly arrived data, but will also provide the updated values for historical data from the master dataset.

In the proposed system, the decision on how to represent the data in the

master dataset has to be made. Marz 2015 in [1] proposes using fact-based model for this purpose. In this model the raw data is stored as atomic facts. It keeps the facts immutable and eternally true, as well as identifiable by using timestamps. Marz also mentions four major benefits of the model:

- **The dataset is queryable at any time in its history**

  The benefit is a direct consequence of facts being timestamped and immutable. Instead of storing only the current state of the world, as one would using a mutable, relational schema, it is possible to query your data for any time covered by your dataset. "Updates" and "deletes" are performed by adding new facts with more recent timestamps, but no data is actually removed, making reconstruction of the data possible. If, for example, we are interested in the number of people a Twitter user is following (is subscribed to his messages), we don't just keep a single record with the information and update it when user starts/stops following people. Instead, a new record is created every time one of these events occurs, adding timestamp information.

- **The data is human-fault tolerant**

  Human-fault tolerance is achieved by simply deleting any erroneous facts. The record is automatically reset by using earlier timestamp. If, for example, user's location is updated by accident, it can be "restored" simply by deleting the last record containing user's location.

- **The dataset easily handles partial information**

  Storing one fact per record makes it easy to handle partial information about an entity without introducing NULL values into the dataset. Suppose that user provides his location, but doesn't provide his age. The dataset will only contain facts for the known information, making any "absent" fact logically equivalent to NULL.

- **The data storage and query processing layers are separate**

  Another key advantage of the fact-based model in Lambda architecture is that the data is kept in both normalized and denormalized (in serving layer) forms, leveraging the benefits of both.

3.3.2.1 Computing on the batch layer

Because the master dataset is continually growing, a strategy for updating batch view has to be reviewed. Marz 2015 in [1] compares two algorithms, that can serve the purpose when new data becomes available. The first one is *recomputation* algorithm, which throws away old batch views and recomputes functions over entire master dataset. A recomputation algorithm is shown in Figure 3.3 on a trivial example of counting the total number of rows in the

dataset. Alternatively, an *incremental* algorithm will compute the function over the new data (calculate the number of rows) and merge it with the old view (add to the number of rows from previous calculation).

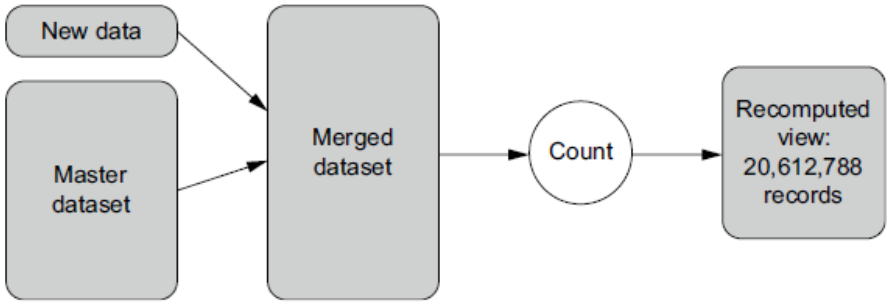

Figure 3.3: A recomputing algorithm to update the number of record in the master data-set [1]

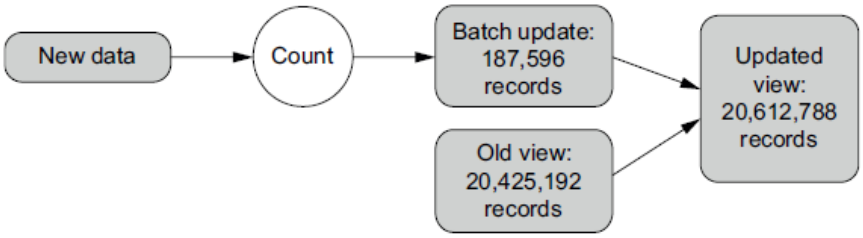

Figure 3.4: An incremental algorithm to update the number of record in the master data-set [1]

On the first glance, it seems that it doesn't make much sense to use a recomputation algorithm, when a more efficient way to achieve the same result exists. However, the efficiency is not the only factor to consider. The key trade-offs between the two approaches are performance, human-fault tolerance, and the generality of the algorithm. [1]

- **Performance**

  There are two aspects to the performance of a batch-layer algorithm: the amount of resources required to update a batch view with new data, and the size of the batch views produced. Incremental algorithm will almost

always use significantly less resources to update a view, as it needs to use new data and the current state of the batch view to perform an update. For a task such as computing the number of rows (as demonstrated in Figure 3.3), the view size will be significantly smaller than the master dataset because of the aggregation. But this is not always the case, as the batch view needs to be formulated in a such a way that it can be incrementally updated. Imagine calculating the number of unique rows instead of a simple row count. To perform an incremental update, not only the current number of unique rows would need to be stored, but also a set of all the unique rows or their indexes.

- **Human-fault tolerance**

  Human mistakes are inevitable in software engineering. Imagine that a bug is introduced with a new deployment, making row count increment by two instead of one for each new row. Correcting such a mistake when using incremental algorithm can be difficult. Overcounted record need to be identified from logs and then correct each of the affected records. Hoping you have the right logs to fix these mistakes is, however, not a good engineering practice. The exact same issue can be corrected easily when using recomputation algorithm by just removing the bug. After the next batch update the batch view will contain correct values, as it is recomputed from scratch.

- **Generality of the algorithm**

  Although incremental algorithms can be faster to run, in some cases they have to be tailored for a specific purpose, as shown in the example with counting unique rows. The problems with the size of the batch view can be tackled by introducing probabilistic counting algorithms, however, such improvements can dramatically increase the complexity of the algorithm and come at cost of the exact result.

As human-fault tolerance is a non-negotiable requirement for a robust data system, the recomputation version of the algorithms is a must have. The system will be designed with this thought in mind. Especially in the environment where continuous business requirements and processing methodology changes occur, implementing recomputation algorithms brings multiple benefits.

3.3.2.2 Storing master dataset

As discussed in data model description, the key property of data is immutability. Each piece of data is written once and is read multiple times. The storage solution must therefore be optimized to handle a large, constantly growing dataset. The batch layer is also responsible for computing functions on the dataset to produce batch views. This means the storage system must be good

at reading lots of data at once. In particular, random access to individual pieces of data is not required, as stated by Marz 2015 in [1]. The author of the architecture also describes distributed filesystems as the perfect candidate for master dataset storage, as it satisfies all the requirements. They are scalable, allow parallel processing and can enforce immutability using permissions systems.

The distributed filesystem will be used in our design as well. However, instead of using classic filesystem like HDFS, we will use one of the filesystems provided as a service by public clouds. Using managed filesystem will highly reduce the complexity connected to patching, deploying and maintaining filesystem. These services also allow scaling easily and save costs, as prices are usually derived from service usage. In our case Amazon Elastic File System (EFS).[29] will be used. Amazon EFS provides scalable, fully managed elastic network filesystem for use in Amazon Web Services (AWS) Cloud. It can be mounted to different AWS services and accessed from virtual machines and containers.

3.3.3 Speed layer

Designing speed layer requires fundamentally different approach than in the case of the batch layer. Low latency becomes the main factor in the speed layer, that's why it is based on incremental computation instead of batch computation. As already discussed in the previous section, incremental computation is significantly more complex than batch computation and brings many challenges. On the other hand, the speed layer usually includes just a fraction of data, making it much smaller compared to batch layer. Because of the complexity, the speed layer can be prone to errors. These errors are, however, short-lived, as they are corrected after the next batch update.

3.3.3.1 Storing real-time views

The obligations of the speed layer views are demanding—the Lambda Architecture requires low-latency random reads, while using incremental algorithms needs low-latency random updates. According to Marz 2015 in [1], the underlying storage layer must therefore meet the following requirements:

- **Random reads**

  A realtime view should support fast random reads to answer query quickly.

- **Random writes**

  In order to support incremental algorithms, it has to be possible to update a real-time view with low latency.

- **Scalability** The real-time views should scale with the read/write rates and the amount of data they store. This typically means that real-time views can be distributed across multiple nodes.

- **Fault tolerance** Real-time views should continue to function even in cases of a disk or a machine crash. The requirement is accomplished by replicating data across multiple nodes so there are backups available.

The mentioned properties are common to a class of databases called *NoSQL databases*. These storages are largely characterized by high-performance reads and wites, usually at the expense of the usual capabilities of a transactional database, the complexity of the queries, or both. [30] The selection of the technology depends on the particular use case. In fact it is not uncommon to combine multiple databases for realtime views to benefit from strenghts of each of them. For example, combining Cassandra store indexes with a key/value format with ElasticSearch for indexes that support search queries.

3.3.3.2 Expiring realtime views

Incremental algorithms and random-write databases increase the complexity of the algorithms and can introduce data inaccuracy. Because the simpler batch layer continuously overrides the speed layer to correct these errors, the speed layer views need to provide only the data not yet processed by the batch layer. Once a batch computation run finishes, a portion of the data from speed layer views can be discarded – but obviously only the data that is now provided by the batch layer views. Everything else must be kept. A generic approach to the problem proposed by Marz 2015 in [1] is to maintain two sets of real-time views and alternate clearing them before each batch layer run, as shown in Figure 3.5. One of those sets (active views) will exactly represent the data required to compensate for the views produced by the batch layer. Before each batch run, inactive views are truncated. After the run is finished, the application reading the data should switch to reading the data from previously truncated views, as the now contain only the data accumulated since start of the previous batch run (data not yet processed by the batch layer).

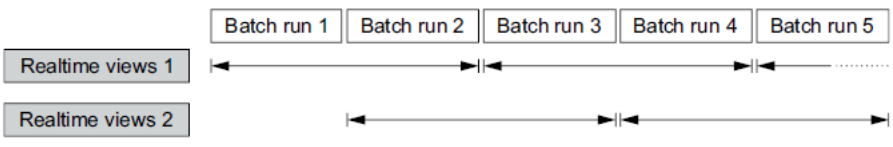

Figure 3.5: Alternating realtime views [1]

It may appear expensive to maintain two real-time views, since it doubles the storages costs, but as the speed layer represents only a tiny portion of our data (usually a few hours), we'll consider this acceptable. The other issue might be imperfect synchronization when truncating one of the views. Either data redundancy or data loss may occur during the synchronization. Since the data will be repaired after the next batch run, we will accept these potential small errors. In cases when these errors are unacceptable, a messsage tagging technique can be applied. Tagging data as soon as it enters the system allows for traceability of when the data entered the system, and thus what corresponding information can be removed.[31]

3.3.4 Serving data

We've already discussed storage requirements for real-time views. But what are the requirements for the views created by the batch layer? It's not surprising that they don't differ much. It needs to be scalable, fault-tolerant and provide quick answers to queries (random reads). Unlike in speed layer, random writes are not required, as the data can be completely swapped after each batch run, making the requirements for the storage less demanding. It has been stated, that different requirement scenarios may suit different NoSQL databases. But what kind of database should we select for our reference application? In our scenario, we need to a fast access to historical data for any of the trending (or rather any) hashtags. In other words, trend lines describing hashtag occurrences in time should be provided with low-latency. For these purposes Apache Cassandra has been selected as it fits these needs really well, which will be discussed in the next section. In the sample application Cassandra will serve views created by both batch and speed layers.

3.3.4.1 Cassandra's data model

First issue that can negatively affect the latency of query in distributed databases are when the values for a queried key is spread across cluster. One of the servers might be under heavy load or be performing garbage collection, causing query response to be slow. Another problem is caused by disk seeks. Ideally, we want to retrieve our data from a single node and the values to be physically stored next to each other on disks. These issues can be addressed by Apache Cassandra, which allows users to decide how the data is partitioned and stored. Cassandra allows to define the *partition key*, which is responsible for data distribution across cluster nodes and *clustering key*, which describes in what order the data should be stored on disks.[32] Both primary and clustering keys can be made by multiple columns. In the sample application we will be querying hashtag time-series data. As illustrated in the Figure 3.6, the data will be partitioned by hashtag, meaning all the data for a hashtag would be possible to retrieve from a single node. We will also order the data

by clustering key, which consists of 'date', 'hour' and 'min_5' (5-minute time slot) columns in descending order, as we don't expect to always query all the data for a hashtag, but just a part of it. At the same time, it is assumed, that the latest data will be demanded more frequently than the old data.

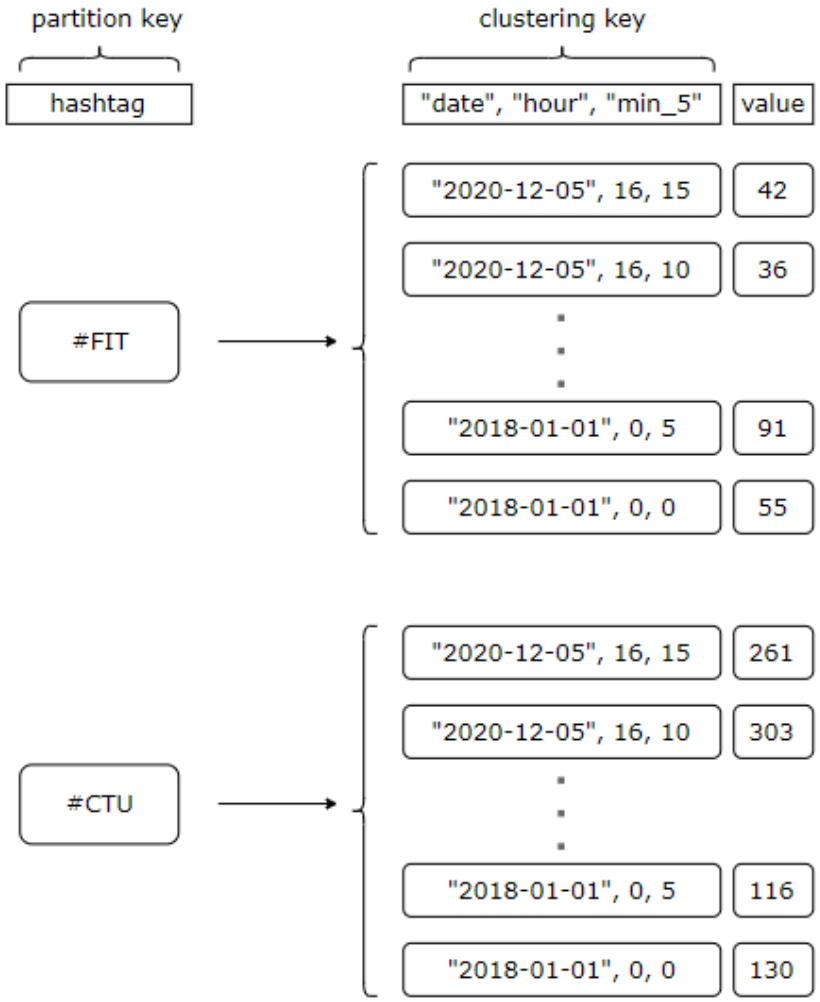

Figure 3.6: Storing time-series data in Cassandra

Besides that, Cassandra offers linear write scalability [33] and there is not a single point of failure in Cassandra cluster, as it uses features of peer-to-peer architecture and uses a gossip protocol to maintain and keep in sync a list of nodes that are alive or dead. No Cassandra nodes perform certain organizing operations distinct from any other node.[32] These characteristics

make Cassandra a perfect match for our requirements.

3.3.5 Data processing

Once we have incoming events feeding our multi-consumer queue, the next step is to process them and update real-time views. At the same time a master dataset needs to be populated by these new events to reflect the new data after the next batch update. One of the major criticisms of Lambda architecture is the necessity to maintain two code bases to produce the same result in two complex distributed systems.[9] Inevitably, code ends up being specifically engineered toward the framework it runs on. We will address this problem by selecting a unified engine that natively supports both batch and streaming workloads. Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. Structured Streaming allows to express the streaming computation the same way it would have been expressed in a batch computation on static data. The Spark SQL engine takes care of running it incrementally and perpetually, and updating the final result as streaming data continually arrives.[34] Using Structured Streaming will allow us to reuse parts of the code between batch and stream processing decrease the complexity of maintaining two code bases.

Structured Streaming uses micro-batched model, which allows higher throughput compared to one-at-a-time processing model. However, this is achieved at cost of higher latency, making it slightly slower[1]. In case of payment processing or fraud detection this latency difference can be crucial, however, we will assume that social media analytics doesn't require performance of stream processing in single-digit milliseconds. Should we prefer the best possible latency against the throughput, Structured Streaming allows to run stream processing in continuous mode (once-at-a-time processing), although this mode is still labeled as experimental.[34]

3.3.6 Application deployment, scaling and management

One of the challenges of big data solutions development is deployment of the complex multi-part software to deploy the software in production systems. To address the challenge, a scalable, reliable and easy to manage platform is required. In the recent years, Kubernetes has become a popular option to deploy application in large-scale infrastructures. Kubernetes allows for comparatively fast provision of a clustered environment, scales on-demand during traffic bursts, improves infrastructure utilization and greatly reduces costs adherent to infrastructure maintenance. Major cloud providers now offer Kubernetes as a Service, including AWS, Google, Microsoft Azure, Oracle and others. [35] This made installing and operating Kubernetes clusters much easier, while keeping vendor lock-in risk low, as containerized application built for Kubernetes can be easily deployed to any other Kubernetes service, regardless

of its underlying infrastructure. Kubernetes also offers lots of addon services like third-party logging or monitoring.

Thanks to scalability and extensibility options, big data applications are good candidates for utilizing the Kubernetes platform. In the recent years more and more of developer groups behind the big data tools adopt Kubernetes support. For example, Apache Spark started supporting Kubernetes as the resource manager. Currently, the Kubernetes scheduler that has been added to Spark is experimental[36], however some major companies like Google begin replacing YARN with Kubernetes.[37]. Running Spark on Kubernetes not only eases and unifies cluster management, but also gives developers the option to isolate jobs. For example, as shown in Figure 3.7, running containerized jobs allows to run multiple Spark applications, which are dependent on different version of Spark.

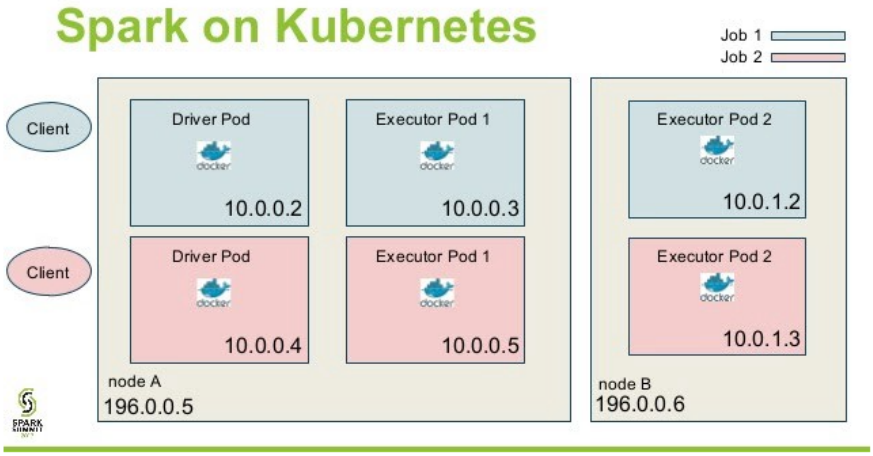

Figure 3.7: Spark on Kubernetes [11]

3.3.7 Delivery mechanism

Once the computed views from batch and speed layers are stored in the Cassandra database, we want to access the data through a web-based interface. In the sample application, a simple dashboard will be created to demonstrate the idea behind the data delivery in the design. Besides delivering the data from Cassandra, it would be a nice feature to be able to keep dashboard data updated without the necessity to constantly request updates by dashboard user.

To fulfill these requirements, we will develop a simple web application, which will serve the data to the client (web browser) and will demonstrate the benefits of using WebSockets. WebSocket protocol enables a full duplex

communication between a server and a client over a long running TCP connection. WebSockets provide an enormous reduction in unnecessary network traffic and latency compared to the unscalable polling solutions.[38] Figure 3.8 compares polling and WebSockets. When using the polling approach, where the client periodically sends requests to the server in order to find out if any updates are available, which becomes highly inefficient. In contrast, in case of WebSockets, after a specific HTTP-based handshake is exchanged between the client and the server, the application-layer protocol is upgraded from HTTP to WebSockets, using the previously established TCP connection. After the upgrade, the data can be sent or received using the Websocket protocol by both endpoints, until the Websocket connection is closed. [12]

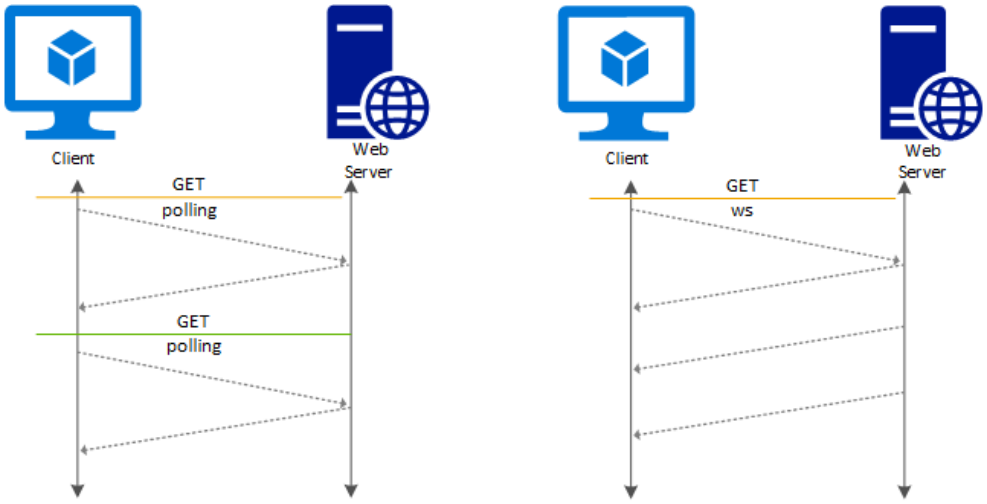

Figure 3.8: Getting server updates through polling and WebSocket [12]

In the sample application we will create a simple service, which will consume updates regarding latest trending hashtags from a Kafka topic and will emit these updates to the clients through WebSockets. On top of that it will serve trendline data from Cassandra for hashtags requested by the client.

3.3.8 Design Summary

In this chapter a system design was proposed to tackle the challenges leading to the fulfilment of system requirements. The design is inspired by a well-established Lambda architecture with some changes introduced to address its complexity drawbacks, utilize modern trends in public cloud computing and to best suit the domain of social media analytics. The proposed system design is summarized in Figure 3.9.

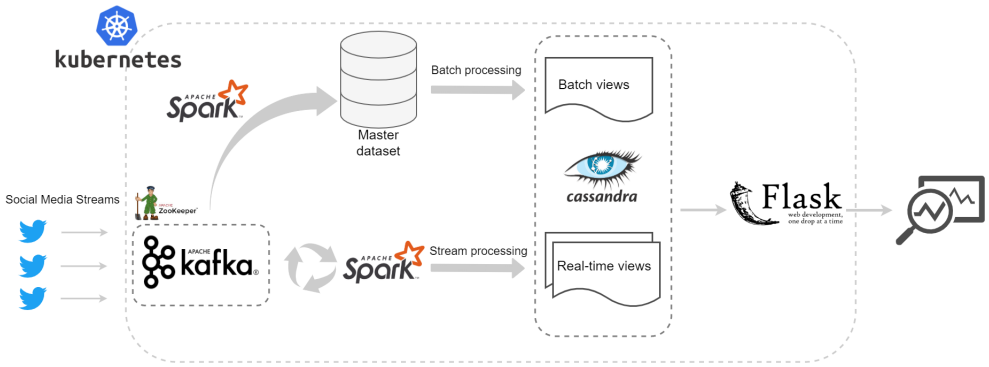

Figure 3.9: System Design

Chapter 4

System Implementation

4.1 Data Collection

4.1.1 Twitter API

Twitter[3] as many other social networks can be accessed via the web or mobile device. Another option provided by Twitter is programmatic access to Twitter data through their application programming interface (API). Twitter limits free API usage, but, compared to other microblogs, they are quite generous. This fact makes Twitter data widely used in scientific researches and data-driven system prototyping. The popularity of the API resulted in creation of many API wrappers – language-specific kits or packages that wrap sets of API calls into easy-to-use functions. One of these wrappers is tweepy [39] – a wrapper for Python, which was used in this implementation. The most interesting API endpoint for our use case allows to access sampled tweet stream. As stated in [40] "The sampled stream endpoint delivers a roughly 1% random sample of publicly available Tweets in real-time. With it, you can identify and track trends, monitor general sentiment, monitor global events, and much more.".

4.1.2 Kafka Producer

Tweet-collector is a Python module created in Python 3.9 to collect tweets using sampled tweet stream utilizing Tweepy library. Besides retrieving tweet stream, the module acts like a Kafka producer, writing newly obtained tweets to a dedicated Apache Kafka topic. Another python library - 'kafka-python' [41] was used as a client for Apache Kafka.

[3]Twitter is an American microblogging and social networking service. More information on https://twitter.com/

4.1.3 Data Serialization

When writing data to Kafka, it is tempting to use a schemeless format like JSON. This approach could quickly backfire, as it may lead to a data corruption. Data corruption issues are hard to debug, because these problems often surface long after the corrupt data is distributed across the system. This may, for example, lead to null pointer exceptions when a mandatory field is missing. It might be easy to realize what the problem is then, but tracing the problem to its source often proves difficult.

Serialization frameworks are an useful tool for making an enforceable schema. Avro[4] supports direct mapping to JSON as well as a compact binary format. It is a very fast serialization framework and support polyglot bindings to many programming languages.[43] Avro schemas are written in JSON. An example of schema for a Tweet object can be seen in Listing 4.1.

```
{
   "type":"record",
   "name":"Tweet",
   "namespace":"cz.cvut.fit.kozlovit.twitterstream",
   "fields":[
      {
         "name":"userid",
         "type":"long",
         "doc":"Id of the user account on Twitter.com"
      },
      {
         "name":"username",
         "type":"string",
         "doc":"Name of the user account on Twitter.com"
      },
      {
         "name":"timestamp",
         "type":"long",
         "logicalType":"timestamp-millis",
         "doc":"Unix epoch time in milliseconds"
      },
      {
         "name":"tweetid",
         "type":"long",
         "doc":"Id of tweet"
      },
      {
         "name":"tweet",
         "type":"string",
         "doc":"The content of the user's Twitter message"
      }
   ],
   "doc":"A basic schema for storing Twitter messages"
}
```

Listing 4.1: Schema definition example

[4] Avro is a row-oriented remote procedure call and data serialization framework developed within Apache's Hadoop project.[42]

4.2 Data processing

4.2.1 Speed layer

The role of speed layer in the system design is to process data streams in real time and fill the gap caused by batch layer's lag in providing views based on the most recent data. The requirement for accuracy is loose, as the results produced by the layer will be eventually replaced by the batch layer. Speed layer was implemented using Structured Streaming processing engine built on Spark SQL – a Spark module for structured data processing. In this project Spark version 3.0.0 was used. Between the programming languages Spark has built-in support for, Scala 2.12 was chosen, since Spark itself was written in Scala and is widely used by data engineers due to superior performance [44] and it suits Spark due to its functional paradigm.

In the sample application the role of the speed layer is quite simple. It consumes records from Apache Kafka topic containing tweets, processes each tweet by extracting hashtags the tweet contains and updates relevant Cassandra speed view records.

4.2.1.1 Consuming data from Kafka

Tweet records are consumed from a dedicated Kafka[5] topic, where Tweet-collectors write new tweets. This makes reading records from Kafka quite easy, as shown in the code example in Listing 4.2.

```
val tweetStream = spark
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers",
      settings.kafka("addressConsumer"))
  .option("subscribe", "tweet_feed")
  .option("minPartitions", settings.kafkaMinPartitions("speedLayer"))
  .option("maxOffsetsPerTrigger",
      settings.maxOffsetsPerTrigger("speedLayer"))
  .load()
```

Listing 4.2: Structured Streaming from Kafka source

Besides specifying stream source, some particularly useful options can be set when reading records from Kafka topic.

The first of the options shown in the code example is 'minPartitions', which allows to specify the desired minimum number of partitions to be read from Kafka and control the parallelism of topic reading.

[5]Kafka is one of the built-in stream sources supported by Structured Streaming and is compatible with Kafka broker versions 0.10.0 or higher.[45].

The second option - 'maxOffsetsPerTrigger' is used to limit the number of offsets processed per trigger interval. Setting this option can be specifically useful when a streaming job is restarted after a crash. Structured streaming uses a mechanism to ensure fault tolerance and correctness in stream processing - checkpointing. Spark does that by saving the metadata of the streaming application to a persistent storage. After the streaming job is restarted after a crash, it will continue right where it left. Without limiting the amount of consumed message per trigger, Spark will try to read all the new messages in the first micro-batch. If the number of messages accumulated during job inactivity rose high enough, the job might potentially crash again due to lack of resources to process these messages at once.

4.2.1.2 Updating speed views

After each micro-batch of records is read from Kafka, it is deserialized from Avro format and transformed using Spark SQL, resulting in a set of records that needs to be stored in the speed views. Since Spark doesn't offer built-in Cassandra sink for Structured Streaming, Spark Cassandra Connector will be used. "This library lets you expose Cassandra tables as Spark RDDs and Datasets/DataFrames, write Spark RDDs and Datasets/DataFrames to Cassandra tables, and execute arbitrary CQL queries in your Spark applications.", as stated in [46].

We will take advantage of Cassandra's special 'counter' column type. A counter column is a column whose value is a 64-bit signed integer and on which 2 operations are supported: incrementing and decrementing. It provides an efficient way to count or sum integer values by using atomic increment/decrement operations on column values[47], which will be used in the speed view tables as shown in Listing 4.3.

```
CREATE TABLE twitter.speed_view_0(
  hashtag text,
  date date,
  hour int,
  min_5 int,
  occurrences counter,
  PRIMARY KEY ((hashtag), date, hour, min_5))
WITH CLUSTERING ORDER BY (date DESC, hour DESC, min_5 DESC);
```

Listing 4.3: Usage of Cassandra counter column

Besides storing data into both speed view tables, hashtags are also written to a dedicated Kafka topic, to be processed by another job to provide hashtag trends.

4.2.2 Hashtag trends

Calculating trending hashtags utilizes window operations available in Structured Streaming (Spark 3.0.0), which allows aggregations over a sliding event-time window. During the aggregations Structured Streaming is also able to handle late-arriving data and update the results even after the sliding window expires. However, the system needs to know when an old aggregate can be dropped from the memory. This is handled by watermarking technique, which lets engine automatically track the current event time in the data and attempt to clean up old state accordingly.[34] By defining threshold, Spark will know how long to keep the data in memory for a potential update. Using windowed operations is demonstrated in the code in Listing 4.4.

```
hashtagStream
  .withWatermark("timtestamp", "20 seconds")
  .groupBy(
    window(col("timestamp"),
      "10 minutes",   // 10 minute windows
      "10 seconds"),  // sliding every 10 seconds
    col("hashtag"))
  .count()
```

Listing 4.4: Window operation

It is worth noting that calculating trending hashtags isn't particularly scalable operation, as it needs a lot of memory to keep temporary results and has to sort hashtags by occurrences. With increasing data volumes, the problem could be tackled by frequency counting algorithms such are Lossy Counting or Sticky Sampling.[48]

4.2.3 Batch layer

As already mentioned, a substantial advantage of Apache Spark is the ability to handle both stream and batch processing. Using the same engine in computation of speed and batch layers allows to reuse codebase and reduces the system complexity. In this project, batch layer was implemented using Spark 3.0.0 and Scala 2.12.12.

The role of a batch layer is to process complete dataset and calculate batch layer views. These views provide exact results, fixing potential errors introduced by the speed layer.

The first requirement before the batch update can be run is that the master dataset is enriched with the data collected since the last batch run. This can be achieved in multiple ways. For example, the data can be continuously streamed as it arrives to a master dataset storage. Another approach is to use pseudo streaming of the data – a technique that brings the streaming feature into the batch job. This approach is shown in the example in Listing 4.5.

```
val query = newTweets
  .writeStream
  .partitionBy("date", "hour")
  .option("checkpointLocation",
      settings.checkpointPaths("masterDatasetSupplier"))
  .option("path", settings.dataPaths("masterDataset"))
  .trigger(Trigger.Once())
  .start()

query.awaitTermination()
```

Listing 4.5: Pseudo streaming job

This feature allows query to fire only once when the job gets triggered. As this approach is technically streaming, checkpoints are created after each trigger, allowing to retrieve all the data accumulated since the last run until now. This way all the new records are appended to the master dataset before every batch run. Partitioning the data, on the other hand, allows efficiently limiting date ranges to be queried.

The data in the master dataset is stored in Parquet[6] format and is partitioned by date and hour to increases the efficiency of ad-hoc querying. Since Apache Parquet is a columnar storage format, meaning the values of each table column are stored next to each other, allowing to skip over non-relevant columns quickly. Besides that, parquet format allows data compression, reducing overall storage costs.

After the newest data is appended to the mater dataset, the whole dataset is processed as a batch job and stored to a Cassandra table. To achieve smooth transition to the newest batch view, we will use a similar approach as in the case of speed views. Two versions of the table will be kept, however, the other case, only one of them will contain any data besides the transition period. Batch job will populate the empty table and when the process is successfully completed, the newly populated table will become 'active' (meaning all the queries for the data from the batch view will be pointed to this table). The second table containing outdated data can now be truncated.

To signal changes when 'active views' are changed, Apache Zookeeper is used. Any service dependent on the data from batch or speed views can get the current information about which table to query from a dedicated *znode*[7].

4.3 Hashtag Dashboard

Hashtag dashboard was implemented to demonstrate how stream processing results can be delivered to the end user. A simple web application was created

[6] Apache Parquet is a columnar storage format of the Apache Hadoop ecosystem [49]

[7] Every node in a ZooKeeper tree is refered to as a znode. Znodes maintain a stat structure that includes version numbers for data changes, acl changes [50]

for this purpose using a client-server model structure.

4.3.1 Application server

The server side of the application was create using Python 3.9 and Flask 1.1.2 – a micro web framework written in Python.[51] It is designed to make getting started quick and easy, with the ability to scale up to complex applications. The simplicity of Flask makes it perfect for such a prototype application to demonstrate a proof of concept.

The goal of the application is to showcase how the system can serve historical data on the fly for a real-time context. The real-time context is represented by the continuously updated list of top trending hashtags that appeared in Tweets during the last 10 minutes. For each of the trendy hashtags (or rather any hashtag, to be precise), the data, describing hashtag's popularity in time, can be retrieved.

Continuous provision of the application outputs to clients is handled using Flask-SocketIO library. It is a Python library that integrates Socket.IO for Flask applications and gives them access to low latency bi-directional communication between the clients and the server. Socket.io is a library that abstracts the WebSocket connections. It also provides a fallback option to communicate with the client if it doesn't support WebSockets.

In order to read the latest updates in hashtag trends produced by the system, the application acts as Kafka consumer, getting the latest updates from a dedicated topic. It acts as a middleman and immediately transmits any updates read from Kafka to the client.

To provide historical data for a hashtag, both batch and speed layer views need to be accessed. The application needs to know which tables to query ('active views') and be able to combine the data. In order to keep track of which tables should be queried to retrieve the correct results, Apache Zookeeper znode data needs to be accessed. Kazoo is a Python library designed to make working with Apache Zookeeper a more hassle-free experience that is less prone to errors.[52] Kazoo enables using Zookeeper watchers and implement application behavior upon znode data changes, which in this case allows the application to duly switch to the right views.

Apache Cassandra allows usage of prepared statements, which are used to retrieve hashtag data from batch and speed views. DataStax Python Driver documentation states that "Prepared statements are queries that are parsed by Cassandra and then saved for later use. When the driver uses a prepared statement, it only needs to send the values of parameters to bind. This lowers network traffic and CPU utilization within Cassandra because Cassandra does not have to re-parse the query each time."[53] After the data from batch and speed views is combined, it is sent to the client in JSON format.

4.3.2 Client application

The client application was crated using HTML, CSS and JavaScript. Plotly.js was used as a charting library, providing rich options to create interactive charts, including 3D charts, statistical graphs, and SVG maps.[54] Socket.IO library was used to utilize WebSocket protocol for receiving data updates from the server. The resulting look of the dashboard can be seen in Figure 4.1.

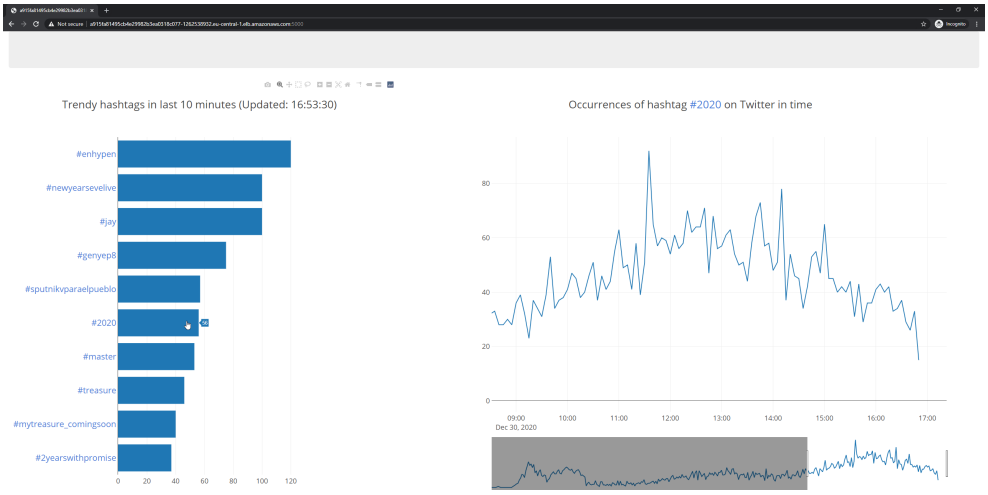

Figure 4.1: Hashtag Dashboard

4.4 Kubernetes Deployment

Using Kubernetes as a platform to manage a stream-processing system workloads and services was a big challenge. Kubernetes offers many benefits and provides a great way to produce prototypes such as the one created in this project in a time and cost-efficient way, allowing portability and scalability in case of the prototype proves successfull. However, although a lot of effort has been made in the last years, the options offered for running big data systems on Kubernetes are limited and not mature enough. One of the major issues concerns limited options for persistent storage that can be shared between different jobs. Spark Kubernetes scheduler is currently still marked as experimental, and other problems need to be addressed. Nonetheless, the open-source community is continuously working on solving these issues in order to make Kubernetes a practical option for deploying big data systems.

4.4.1 Kubernetes cluster

A Kubernetes cluster can be deployed on either physical or virtual machines. Minikube offers a lightweight implementation that creates a virtual machine and deploys a simple cluster containing one node.[55] Minikube provided a great option in the initial steps of the development, allowing experimenting in the local environment. It allowed to test the deployment of individual system components and test integration of the related component groups. Local machine resources, however, did not permit the deployment of a whole system in the Minikube environment.

After exceeding local resources limits, the services of a public cloud provider were used to deploy and test the system with all of its components. For this project Amazon Web Services (AWS) were used. Amazon Web Services offers reliable and scalable cloud computing services and offer extensive service list.[56] One of these services is Amazon Elastic Kubernetes Service (Amazon EKS) – a managed Kubernetes cluster provided as a service.[57] Amazon EKS gives a user the flexibility to start, run and scale Kubernetes applications in the AWS cloud. For creating and deleting clusters on EKS eksctl was used. Eksctl is a simple command line interface tool for creating clusters on EKS.[58]. The system was deployed on a cluster consisting of 5 *t3a.large*[8] instances, totaling in 10 2.5GHz virtual processing units (vCPU) and 40GB RAM memory.

4.4.2 Deployment strategy

The basic application deployment on Kubernetes makes use of kubectl – the Kubernetes command-line tool that allows to run commands against Kubernetes clusters. Besides deploying applications, it can be used to inspect and manage cluster resources or view logs. To create, modify and delete Kubernetes resources, Kubernetes manifests are used. These manifests contain the desired state of the application, that Kubernetes will maintain when the manifest is applied. Using kubectl allows to apply manifests stored in local files, usually in a YAML format. An example of deployment described in YAML format is shown in Listing4.6.

[8]Instance type provided by Amazon Web Services [59]

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hashtag-dashboard-deployment
  labels:
    app: hashtag-dashboard
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hashtag-dashboard
  template:
    metadata:
      labels:
        app: hashtag-dashboard
    spec:
      containers:
      - name: hashtag-dashboard
        image: dockerforgori/hashtag_dashboard:1.0.6
        ports:
        - containerPort: 5000
        envFrom:
        - configMapRef: { name: twitter-streaming-config }
        - secretRef: { name: twitter-streaming-secret }
```

Listing 4.6: Spark Application deployment

The example shows hashtag dashboard deployment. The application was containerized using Docker[9] platform and hosted on Docker Hub – a docker image repository, from which the application image is pulled. The same deployment strategy was used in the case of tweet Tweet collector.

In the case of more complex projects, when a number of manifests might grow to dozens or even hundreds, organizing them might become a difficult task. One of the possible solutions to address the problem might be Helm – the package manager for Kubernets using packaging format called charts. Helm charts are easy to create, version, share and publish.[61] Helm charts are, essentially, collections of preconfigured application resources that can be deployed as one unit, while providing an easy option for reconfigurations and customizations. Some of widely used applications can be found in helm chart repositories like in the one maintained by Bitnami.[62] In this project Apache Cassandra and Apache Kafka were installed using Helm.

Another option for packaging, deploying, and managing a Kubernetes application are Kubernetes operators. Operators are software extensions to Kubernetes that make use of custom resources to manage applications and their components.[55] They extend the functionality of the Kubernetes API to create, configure and manages applications on behalf of a Kubernetes user. This is particularly useful in the case of complex applications like Spark, therefore Kubernetes Operator for Apache Spark ([63]) was used in this project.

[9]Docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers.[60]

4.4.3 Spark Operator

Kubernetes Operator for Apache Spark leverages the operator pattern for managing the life cycle of Spark applications on a Kubernetes cluster. The way life cycles of Spark applications are managed (how applications get submitted to run on Kubernetes and how application status is tracked), is vastly different from that of other types of workloads on Kubernetes (e.g., Deployments). The Kubernetes Operator for Apache Spark reduces the gap and allows Spark applications to be specified, run, and monitored idiomatically on Kubernetes.[63] The operator allows Spark applications to be specified using YAML files and run without the need to deal with the spark-submit[10] process. The process of spark-submit can be seen in Figure 4.2 After a SparkApplication (a custom Kubernetes resource defined by Spark Operator) object, is created using kubectl command and provided manifest, it is recognized and picked by the Spark Operator. The operator then submits the application using spark-submit, providing the specified arguments and creates the driver pod of the application. After the driver pod is started, it creates executor pods. Spark operator then continues to monitor the pods and update their status.

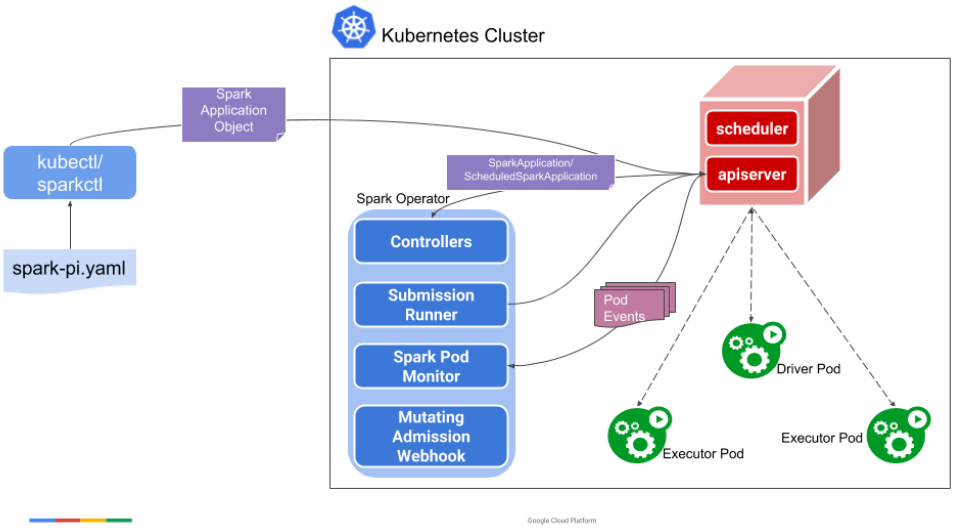

Figure 4.2: Spark operator architecture [13]

Spark operator allows specifying Spark applications in a declarative manner. An example of 'SparkApplication' resource is shown in Listing 4.7. The

[10]spark-submit is a script that takes care of setting up the classpath with Spark and its dependencies

operator also supports 'ScheduledSparkApplication' resource type, which was used to schedule batch updates.

```
apiVersion: "sparkoperator.k8s.io/v1beta2"
kind: SparkApplication
metadata:
  name: speed-layer
  namespace: default
spec:
  type: Scala
  mode: cluster
  image: "dockerforgori/spark3_twitter:1.0.3"
  imagePullPolicy: Always
  mainClass: cz.cvut.fit.kozlovit.twitterstream.SpeedLayer
  mainApplicationFile: "local:///opt/spark/jars/twitter-processing_2.12-3.0.0_1.0.jar"
  sparkVersion: "3.0.0"
  restartPolicy:
    type: OnFailure
    onFailureRetries: 3
    onFailureRetryInterval: 10
    onSubmissionFailureRetries: 3
    onSubmissionFailureRetryInterval: 10
  volumes:
  - name: persistent-storage
    persistentVolumeClaim:
      claimName: efs-claim
  driver:
    cores: 1
    memory: "512m"
    serviceAccount: spark
    volumeMounts:
      - name: persistent-storage
        mountPath: /data
    javaOptions: "-Dconfig.resource=test.conf"
    envFrom:
    - configMapRef: { name: twitter-streaming-config }
    - secretRef: { name: twitter-streaming-secret }
  executor:
    cores: 1
    instances: 1
    memory: "2g"
    volumeMounts:
      - name: persistent-storage
        mountPath: /data
    javaOptions: "-Dconfig.resource=test.conf"
    envFrom:
    - configMapRef: { name: twitter-streaming-config }
    - secretRef: { name: twitter-streaming-secret }
```

Listing 4.7: Spark Application deployment

4.4.4 Persistent storage

One of the challenges of running big data systems on Kubernetes are persistent storage options. This turned out to be true in our case of streaming jobs and checkpointing. It has been shown that Spark batch processing works well over S3 – an AWS object storage service as shown in [64]. However, this was not the case for Structured streaming applications as discussed in [65]. Since S3 storage is eventually consistent [66], Spark tasks might start failing with a "directory not found" error due to S3s read-after-write semantics. Another service AWS provides are Elastic File System. It is a distributed file system which can be mounted into pods running Spark applications, tackling the

problem. Another benefit of the service is that the data persists even when the cluster is shut down, allowing cost savings by on-demand resource usage without losing the data. EFS was used as a persistent storage solution for both streaming and batch processing applications.

4.5 System Monitoring

Kubernetes allows to deploy a web-based user interface (Dashboard). Dashboard provides information on the state of Kubernetes resources in the cluster and on any errors that may have occurred.[55] Dashboard was especially useful for tuning Spark jobs by tracking the resource usage. Dashboard also allows to display logs from containers or even deploying containerized applications through a wizard. The look of the Kubernetes dashboard can be seen in Figure 4.3.

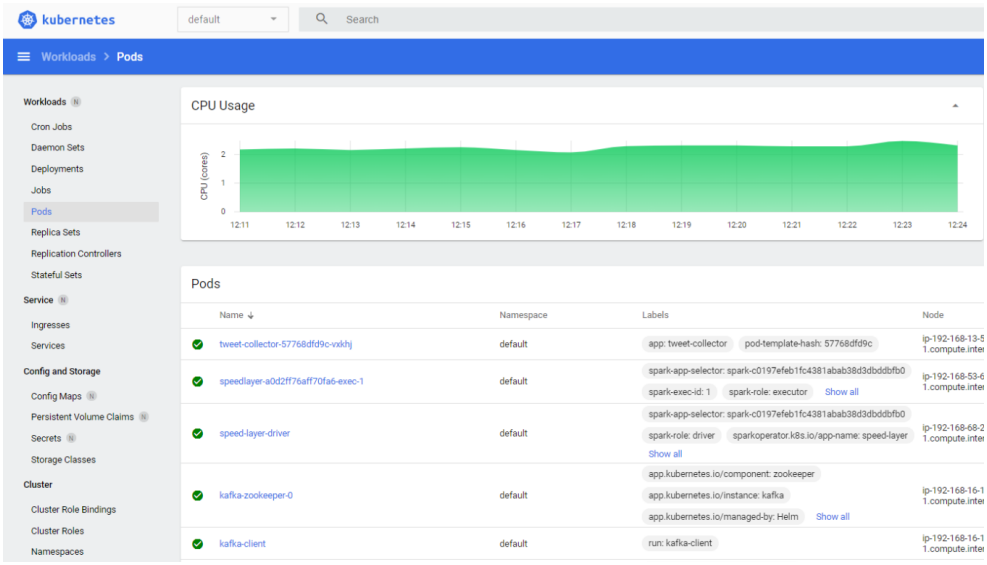

Figure 4.3: Kubernetes Dashboard

Chapter 5

System Evaluation

5.1 Test Scenarios

In order to evaluate the performance of the system prototype, we will design test scenarios, which will help us understand and evaluate how well the prototype meets its key requirements. Test scenarios were primarily created to test the main part of the system, which means we will be testing Apache Kafka - Spark Structured Streaming - Apache Cassandra pipeline. In these tests the stream job reads Kafka topic containing tweets, processes the records (extracts hashtags from them) and stores the results to Cassandra table.

The purpose of the tests is to get insights about the processing latency, throughput and how dependent these metrics are. Besides that, we want to test system fault tolerance and see if the system's performance is affected by the amount of available resources.

The tests are run on the system prototype deployed on Amazon Elastic Kubernetes Service cluster. The two cluster configurations used for the scenarios can be seen in Table 5.1.

	dev	**test**
Kafka nodes	1	3
Kafka replication factor	1	3
Cassandra nodes	1	3
Cassandra replication factor	1	3
Spark driver cores	1	1
Spark driver memory (GB)	1	1
Spark executor instances	1	3
Spark executor cores	1	1
Spark executor memory (GB)	2	2

Table 5.1: Cluster testing configurations

5.1.1 Latency

In the first scenario we will examine how the system reacts to a load change. For this purpose, Kafka producer was set-up to write 1000 generated tweets per second into the topic read by the streaming job. After a while, the second Kafka producer starts to write in the same Kafka topic, generating 6000 tweets per second.

The metrics provided by the Spark UI, showing the impact of the load change can be seen in Figure 5.1.

Streaming job is able to increase its process rate when the input rate increases. However, the result is achieved by increasing the number of records processed in a single micro-batch (input rows), which negatively affects latency (batch duration).

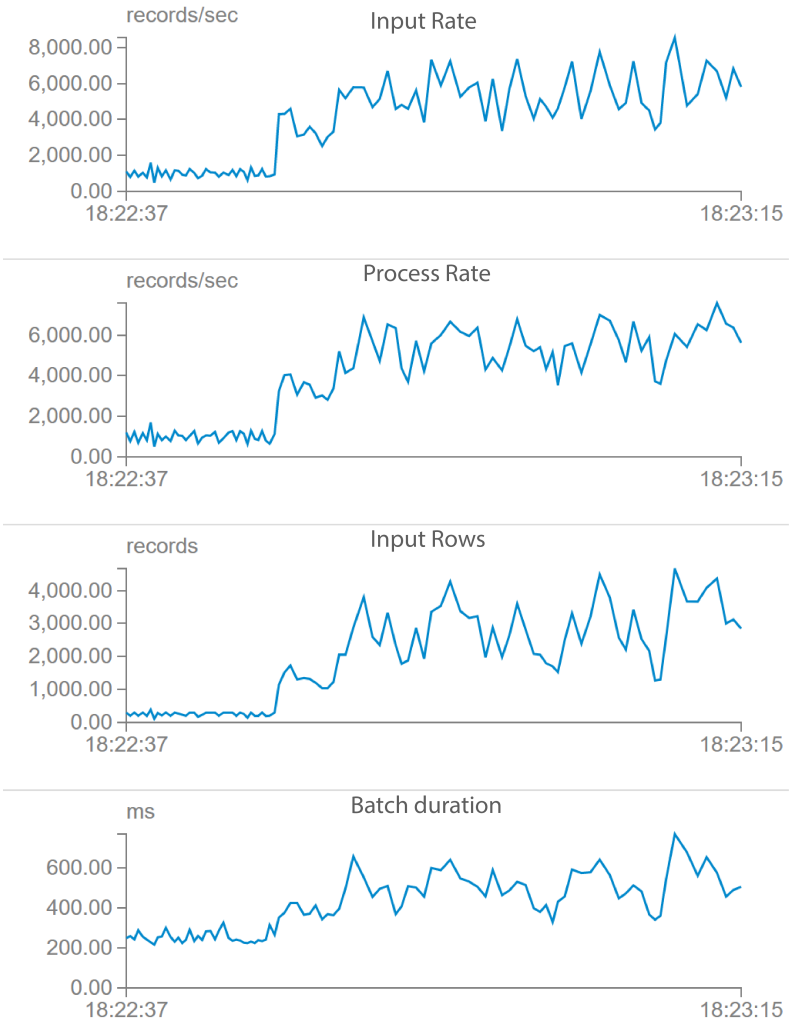

Figure 5.1: The impact of load change on batch duration

5.1.2 Throughput and Scalability

As seen from the previous scenario, the increased load affected the micro-batch size, which increased the latency. In the second scenario we will set fixed batch size when reading from the beginning of Kafka topic containing over 5.1 million tweets to see the system throughput given the batch size. After all the tweets are processed, average process rate and batch duration will be evaluated.

Table 5.2, confirms result of the previous test scenario. Increasing batch size causes higher process rates, but at the cost of an increased batch duration.

The figure also demonstrates that the system performed much better using the 'test' configuration, showing higher process rates and lower latencies for the same batch size compared to the 'dev' configuration, indicating system scalability.

Another interesting finding concerns two of the biggest batch sizes, where no visible difference in process rate can be seen, suggesting the process rate limit for the particular configuration has been hit.

Cluster	Batch Size (records)	Process Rate (records/second)	Batch Duration (ms)
dev	1000	3500	290
	10 000	9800	1020
	30 000	12 400	2420
	50 000	12 300	4070
test	1000	4100	240
	10 000	16 500	610
	30 000	22 200	1350
	50 000	22 800	2200

Table 5.2: The effect of batch size on process rate and batch duration

5.1.3 Fault tolerance

To test the fault tolerance, we will examine the behavior of the application in the case when the application driver pod is suddenly killed, simulating unexpected error. Without the driver pod, the executor pod is terminated as well. Thanks to the Kubernetes self-healing mechanism, the driver pod is recreated, as well as the new executor after a request from the driver. Thanks to the checkpointing mechanism, the streaming job can continue right where it was stopped.

Figure 5.2 shows the scenario, in which the executor pod is killed. The processing of records ceases after the executor dies, but after it is recreated, it processes all the records accumulated in Kafka topic during the respawn

period as one relatively big batch and then continues processing records at the same rate, they were processed before the simulated error.

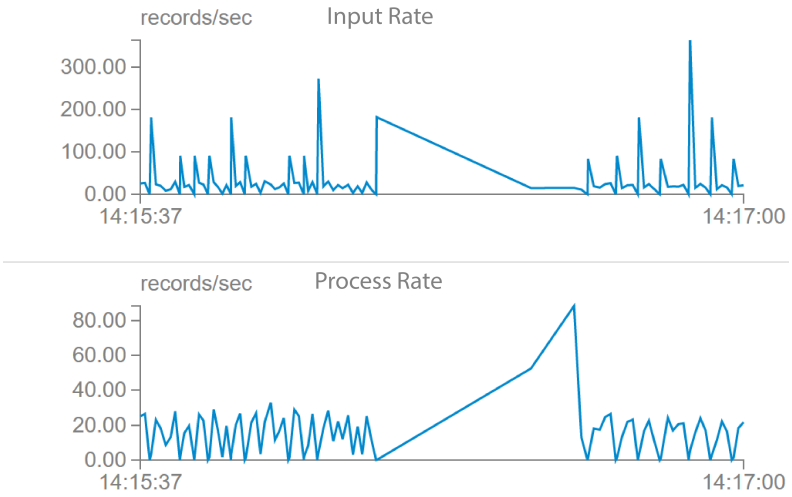

Figure 5.2: Spark executor recovery

5.2 Evaluation of system requirements

5.2.1 Functional requirements

FR1 - Efficient data storage

The system designe proposes using fact-based data model. The master dataset, where all the data is stored, is immutable and supports only read and append operations. This brings simplicity and human-fault tolerance to the system, preventing the data loss caused by mistakes made by operators.

To solve the problem of persistent storage, the system utilizes using shared file systems available as a service by public cloud providers. The sample application uses Amazon Elastic File System, where the master dataset is stored. The service allows dynamic scaling, reducing the storage costs. The data is stored using Apache Parquet columnar format, which allows data compression and efficient ad-hoc querying.

FR2 - Fast access to historical data for real-time context

To provide historical data on the fly, the system utilizes precomputed views created by batch and speed layers. The views are stored in the database system selected to suit the particular use case for the best performance. This was demonstrated in the sample application, which takes

advantage of Cassandra's data model. It allows storing data that are expected to be demanded in a single query on the same node of Cassandra cluster. At the same time, it can be defined how the data should be sorted on disks, which is a great suit for time-series data.

FR3 - Datasets are queryable at any time in their history

Using the philosophy presented in Lambda architecture, all the historical data is kept in the master dataset. Fact-based model assures that no data is lost by update and delete operations, which are not permitted.

FR4 - Ad hoc queries support

The master dataset is stored on a shared file system, where all the data is conveniently available in one location. The file system can be mounted from machines provided by AWS or even from Kubernetes pods. Applying file system permissions prevents data loss or corruption when accessed. This enables ad hoc consumers to access the data and query it according to their needs.

FR5 - Addable data sources and formats

The system proposes a general approach to data ingestion, processing and delivery. It is suited to handle various data sources and formats with only minimal changes in data collection and processing logic.

FR6 - Building new views

Building new views is easy, since it only requires defining new logic to process the master dataset. The new view will not only contain the data accumulated after the view definition, but all the historical data as well.

FR7 - Adaptive views

Views can also be updated, since it essentially means discarding the old view and defining the new one with the updated definitions, which project new business requirements or processing techniques. Batches are automatically rerun and the updated data is promptly available.

FR8 - Delivery mechanism

The proposed system for the data delivery is demonstrated in the sample application. A simple web application was implemented to showcase continuous data updates on the client initiated from the server. The data is efficiently transmitted using WebSocket with a minimal delay after reading updates from Apache Kafka.

The application also demonstrates how the data from batch and speed layers can be combined, serving time-series data covering both the oldest available data in the system with the data produced by external systems seconds ago.

5.2.2 Non-functional requirements

NFR1 - Scalability

The system was designed and implemented with scalability in mind mainly by utilizing distributed subsystems. The incoming data is queued in Apache Kafka, a highly scalable messaging system. The data is then processed by batch and speed layers, both implemented using Apache Spark - a distributed cluster computing framework. Apache Cassandra then provides linear scalability for data storage. The whole system is deployed on Kubernetes, making it easy to provide additional resources if required. Testing the speed layer showed that increased resources indeed had a significant performance effect.

NFR2 - Low-latency processing

Although Structured Streaming's micro-batch processing is more throughput rather than latency-oriented, the tests showed it is able to maintain low-latency even under heavy loads. This metric, however, is highly dependent on the nature of the process.

NFR3 – Fault tolerance

The system offers multiple ways to assure fault-tolerance. The first aspect is human-fault tolerance, which can cause data corruption or loss. This issue is addressed by the fact-based data model and immutability of the master dataset.

The second aspect is represented by hardware failures and is solved using replication and recovery mechanisms. Apache Kafka and Apache Spark replicate their respective data across the nodes, so losing one of the nodes of either of the systems should not affect the overall system functionality.

As for data processing, Structured Streaming uses checkpointing technique, giving Spark application the ability to proceed with streaming jobs right where it left when either job driver or executor encounters failure, as shown in tests. Kubernetes offers system monitoring and self-healing by restarting failed containers.

NFR4 - Generalization

The system can support a wide range of applications. It is inspired by Lambda architecture, which proved itself across many industries, meaning the system can be adjusted to handle various scenarios concerning system specific requirements.

NFR5 - Extensibility

Because the master dataset can contain arbitrary data, adding new types of data can be achieved easily. Adding new views and tweaking the old ones is supported as well.

NFR6 - Cloud platform compatible

The system utilizes the benefits provided by the Kubernetes platform, which can be run on bare metal servers, virtual machines, as well as on private or public clouds. To demonstrate this, the sample application was successfully deployed on Amazon Web Services.

5.3 Future work and opportunities

5.3.1 Auto-scaling options

It's been shown that the system was designed to be scalable and can handle different volumes of data by utilizing more resources. The next step for the system is to use the full power of Kubernetes and take advantage of dynamic autoscaling.

On the app-level Spark gives the ability to the application to request more executors at runtime. Based on the workload, new executor pods can be either created or deleted to achieve a desired performance.

On a cluster-level, Kubernetes allows to add more nodes to the cluster if more capacity to schedule pods is needed, or remove them if the nodes become unused.

In combination, the described autoscaling options allow to handle traffic bursts without the need to of reconfiguration, utilizing cluster resources in a cost-effective manner.

5.3.2 Job scheduling

An important part of the proposed hybrid system is the batch layer, which regularly updates views to provide historical data. In the system prototype, a simple cron job is used to schedule these updates. This solution, however, will not be sufficient in cases when more complex, multi-step processing is expected. This problem can be tackled by using one of the workflow management platforms like Apache Airflow[67].

Airflow treats a collection of tasks as directed acyclic graph (DAG), reflecting tasks relationships and prerequisites. This allows Airflow to run non-dependent tasks in parallel, starting tasks only when all its prerequisites are successfully completed or configuring fallback options in the case of a failure.

5.3.3 Advanced monitoring

Monitoring of a big data system is just as important as the system itself. Monitoring allows for a problem detection, proactive response and overall good health of a computer system. It allows the system to be more stable and reliable.

Some popular tools have emerged in the recent years, which proved to be helpful in monitoring Kubernetes cluster. One of these tools is Prometheus[68] Prometheus allows to regularly collect and store metrics from various applications such as databases or messaging systems. Prometheus offers many ready-to-use exporters for many of the widely used tools including Apache Kafka and Cassandra.

Visualizing metrics collected by Prometheus is easy with Grafana[69]. Grafana is analytics and interactive visualization web application. It allows efficiently query, display and analyze the data. Grafana comes with alerting system, allowing timely notifications in case of emerging system anomalies.

Conclusion

The main aim of the thesis was to design a stream processing system with a focus on high availability, low latency and horizontal scalability.

Key concepts connected to large scale data processing were explored, specifics of processing data streams were analyzed - along with the best practices and pitfalls of designing stream processing systems. Subsequently, a system was proposed. The main strengths of the system were discussed in detail and arguments on why it fits the system requirements provided.

Besides utilizing strengths of well-established design patterns in stream processing systems, the system takes advantage of some of the newest trends and technology in the field. In some cases, this meant relying on experimental software and tackling a lack of well founded resources.

As a proof of concept, a cloud-based system prototype was successfully implemented. The system was deployed and tested on one of the public clouds.[11] Thanks to the system design it can be deployed to different environments with only some minor adjustments.

The system design offers many opportunities for the future, as discussed in the last chapter. The technology (or rather a combination of multiple technologies) used in the implementation is predicted to replace some current standards, making this thesis potentially beneficial in the future.

[11]The repository containing sample application source code is available on https://gitlab.fit.cvut.cz/kozlovit/ni-dip-project-kozlovit

Bibliography

[1] Marz, N.; Warren, J. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., first edition, 2015, ISBN 978-1617290343.

[2] White, T. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., fourth edition, 2015, ISBN 978-1-491-90163-2.

[3] Spark Cluster Mode Overview. [online], [cit. 2020-12-02]. Available from: `https://spark.apache.org/docs/latest/cluster-overview.html`

[4] Luksa, M. *Kubernetes in Action*. Manning Publications Co., first edition, 2018, ISBN 9781617293726.

[5] Kleppmann, M. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, Inc., first edition, 2017, ISBN 978-1-449-37332-0.

[6] Big data architecture style. [online], [cit. 2020-12-20]. Available from: `https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/big-data`

[7] Lambda Architecture. [cit. 2020-10-08]. Available from: `http://lambda-architecture.net/`

[8] Marz, N. How to beat the CAP theorem. [online], 2011, [cit. 2020-10-08]. Available from: `http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html`

[9] Kreps, J. Questioning the Lambda Architecture. [online], 2014, [cit. 2020-10-08]. Available from: `https://www.oreilly.com/radar/questioning-the-lambda-architecture/`

[10] Twitter Trends. [online], 2020, [cit. 2020-12-27]. Available from: `https://twitter.com/`

[11] Kubernetes and Big Data: A Gentle Introductione. [online], [cit. 2020-12-15]. Available from: https://medium.com/sfu-cspmp/kubernetes-and-big-data-a-gentle-introduction-6f32b5570770

[12] Overview of WebSocket support in Application Gateway. [online], [cit. 2020-12-20]. Available from: https://docs.microsoft.com/en-us/azure/application-gateway/application-gateway-websocket

[13] Spark Operator Architecture. [online], [cit. 2020-12-11]. Available from: https://github.com/GoogleCloudPlatform/spark-on-k8s-operator/blob/master/docs/design.md

[14] Yarabarla, S. *Learning Apache Cassandra*. Packt Publishing Ltd., second edition, 2017, ISBN 978-1-78712-729-6.

[15] Saxena, S.; Gupta, S. *Practical Real-Time Data Processing and Analytics*. Packt Publishing Ltd., first edition, 2017, ISBN 978-1-78728-120-2.

[16] Apache Hadoop Website. [online], [cit. 2021-01-02]. Available from: https://hadoop.apache.org/

[17] Zaharia, M.; Chowdhury, M.; et al. Spark: Cluster Computing with Working Sets. *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, 07 2010: pp. 10–10.

[18] Chambers, B.; Zaharia, M. *Spark: The Definitive Guide*. O'Reilly Media, Inc., first edition, 2018, ISBN 978-1-491-91221-8.

[19] Spark Release 2.3.0. [online], [cit. 2020-12-02]. Available from: https://spark.apache.org/releases/spark-release-2-3-0.html

[20] The rise of Kubernetes epitomizes the transition from big data to flexible data. [online], [cit. 2020-12-02]. Available from: https://www.zdnet.com/article/the-rise-of-kubernetes-epitomizes-the-move-from-big-data-to-flexible-data/

[21] Containers 101: What are containers. [online], [cit. 2020-12-19]. Available from: https://cloud.google.com/containers

[22] What is Streaming Data? [online], [cit. 2020-12-27]. Available from: https://www.confluent.io/learn/data-streaming/

[23] Mass, G.; Garillot, F. *Stream Processing with Apache Spark: Best Practices for Scaling and Optimizing Apache Spark*. O'Reilly Media, Inc., first edition, 2019, ISBN 978-1-491-94417-2.

[24] Kalipe, G.; Behera, R. Big Data Architectures : A detailed and application oriented review. 10 2019.

[25] Iqbal, M. Twitter Revenue and Usage Statistics (2020). [online], 2020, [cit. 2020-12-28]. Available from: https://www.businessofapps.com/data/twitter-statistics/

[26] Joyce, B.; Deng, J. Sentiment analysis of tweets for the 2016 US presidential election. In *2017 IEEE MIT Undergraduate Research Technology Conference (URTC)*, 2017, pp. 1–4, doi:10.1109/URTC.2017.8284176.

[27] Why run Apache Kafka on Kubernetes? [online], [cit. 2020-12-28]. Available from: https://www.redhat.com/en/topics/integration/why-run-apache-kafka-on-kubernetes

[28] Stopford, B. *Designing Event-Driven Systems*. O'Reilly Media, Inc., first edition, 2018, ISBN 978-1-492-03824-5.

[29] Amazon Elastic File System. [online], [cit. 2020-12-23]. Available from: https://aws.amazon.com/efs/

[30] Ellis, B. *Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data*. John Wiley and Sons, Inc., first edition, 2014, ISBN 978-1-118-83791-7.

[31] VANHOVE, T.; SEGHBROECK, G. V.; et al. Managing the Synchronization in the Lambda Architecture for Optimized Big Data Analysis. *IEICE Transactions on Communications*, volume E99.B, no. 2, 2016: pp. 297–306, doi:10.1587/transcom.2015ITI0001.

[32] Carpenter, J.; Hewitt, E. *Cassandra: The Definitive Guide: Distributed Data at Web Scale*. O'Reilly Media, Inc., third edition, 2020, ISBN 978-1-098-11509-8.

[33] Benchmarking Cassandra Scalability on AWS — Over a million writes per second. [online], [cit. 2020-12-02]. Available from: https://netflixtechblog.com/benchmarking-cassandra-scalability-on-aws-over-a-million-writes-per-second-39f45f066c9e

[34] Structured Streaming Programming Guide. [online], [cit. 2020-12-02]. Available from: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

[35] The future of big data is Kubernetes. [online], [cit. 2020-12-11]. Available from: https://banzaicloud.com/blog/future-of-big-data-is-kubernetes/

[36] Running Spark on Kubernetes. [online], [cit. 2020-12-15]. Available from: https://spark.apache.org/docs/latest/running-on-kubernetes.html/

[37] Big Data: Google Replaces YARN with Kubernetes to Schedule Apache Spark. [online], [cit. 2020-12-15]. Available from: https://thenewstack.io/big-data-google-replaces-yarn-with-kubernetes-to-schedule-apache-spark/

[38] About HTML5 WebSockety. [online], [cit. 2020-12-20]. Available from: https://www.websocket.org/aboutwebsocket.html

[39] Tweepy: Twitter for Python! [online], [cit. 2020-12-20]. Available from: https://github.com/tweepy/tweepy

[40] Sampled stream. [online], [cit. 2020-12-20]. Available from: https://developer.twitter.com/en/docs/twitter-api/tweets/sampled-stream/introductionl

[41] Kafka Python client. [online], [cit. 2020-12-20]. Available from: https://github.com/dpkp/kafka-python

[42] Apache Avro website. [online], [cit. 2020-12-19]. Available from: https://avro.apache.org/

[43] Using Avro for Big Data and Data Streaming Architectures: An Introductiont. [online], [cit. 2020-12-20]. Available from: https://dzone.com/articles/avro-introduction-for-big-data-and-data-streaming

[44] Why learn Scala Programming for Apache Spark. [online], [cit. 2020-12-20]. Available from: https://www.dezyre.com/article/why-learn-scala-programming-for-apache-spark

[45] Structured Streaming + Kafka Integration Guide (Kafka broker version 0.10.0 or higher). [online], [cit. 2020-12-10]. Available from: https://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html

[46] Spark Cassandra Connector. [online], [cit. 2020-10-08]. Available from: https://github.com/datastax/spark-cassandra-connector

[47] http://clojurecassandra.info/articles/cql.html. [online], [cit. 2020-12-06]. Available from: http://clojurecassandra.info/articles/cql.html

[48] Frequency Counting Algorithms over Data Streams. [online], [cit. 2020-12-15]. Available from: https://micvog.com/2015/07/18/frequency-counting-algorithms-over-data-streams/

[49] Apache Parquet Website. [online], [cit. 2020-12-19]. Available from: https://parquet.apache.org/

[50] Zookeeper documentation. [online], [cit. 2020-12-19]. Available from: https://zookeeper.apache.org/doc/r3.1.2/zookeeperProgrammers.html

[51] Flask. [online], [cit. 2020-12-20]. Available from: https://github.com/pallets/flask/

[52] Kazoo. [online], [cit. 2020-12-11]. Available from: https://github.com/python-zk/kazoo

[53] DataStax Python Driver 3.24. [online], [cit. 2020-12-27]. Available from: https://docs.datastax.com/en/developer/python-driver/3.24/getting_started/

[54] Plotly JavaScript Open Source Graphing Library. [online], [cit. 2020-12-11]. Available from: https://plotly.com/javascript/

[55] Kubernetes documentation. [online], [cit. 2020-12-27]. Available from: https://kubernetes.io/docs/

[56] Amazon Wev Services website. [online], [cit. 2020-12-27]. Available from: https://aws.amazon.com/

[57] Amazon Elastic Kubernetes Service. [online], [cit. 2020-12-27]. Available from: https://aws.amazon.com/eks/

[58] eksctl - The official CLI for Amazon EKS. [online], [cit. 2020-12-19]. Available from: https://github.com/weaveworks/eksctl

[59] Amazon EC2 Instance Types. [online], [cit. 2020-12-19]. Available from: https://aws.amazon.com/ec2/instance-types/

[60] Docker website. [online], [cit. 2020-12-19]. Available from: https://www.docker.com/

[61] Helm website. [online], [cit. 2020-12-27]. Available from: https://helm.sh/

[62] Bitnami Charts. [online], [cit. 2020-12-27]. Available from: https://bitnami.com/

[63] Spark Operato. [online], [cit. 2020-12-11]. Available from: https://github.com/GoogleCloudPlatform/spark-on-k8s-operator

[64] Top 5 Reasons for Choosing S3 over HDFS. [online], [cit. 2020-12-16]. Available from: https://databricks.com/blog/2017/05/31/top-5-reasons-for-choosing-s3-over-hdfs.html

[65] Improving Spark Streaming Checkpointing Performance With AWS EFS. [online], [cit. 2020-12-02]. Available from: https://blog.yuvalitzchakov.com/improving-spark-streaming-checkpoint-performance-with-aws-efs/

[66] Introduction to Amazon S3. [online], [cit. 2020-12-19]. Available from: https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.htmll

[67] Apache Airflow Website. [online], [cit. 2020-12-22]. Available from: https://airflow.apache.org/

[68] Prometheus Website. [online], [cit. 2020-12-19]. Available from: https://prometheus.io/

[69] Grafana website. [online], [cit. 2020-12-19]. Available from: https://grafana.com/

Appendix A

Acronyms

API Application Programming Interface

AWS Amazon Web Services

CAP Consistency, Availability, Partition Tolerance

CEP Complex Event Processing

CPU Central Processing Unit

CQL Cassandra Query Language

CSS Cascading Style Sheets

DAG Directed Acyclic Graph

DataOPS Data Operations

DevOps Development Operations

EFS Elastic File System

EKS Elastic Kubernetes Service

HDFS Hadoop Distributed File System

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notation

MAU Monthly Active Users

OS Operating System

QA Quality Assurance

RDD Resilient Distributed Datasets

SQL Structured Query Language

SVQ Scalable Vector Graphics

TCP Transmission Control Protocol

VMs Virtual Machines

YAML Ain't Markup Language

YARN Yet Another Resources Negotiator