University of Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation: Computer Science	Spring semester, 2019 Open/~~Confidential~~
Author: Nicolai Vikanes Stensland	…… (signature of author)

Faculty supervisor:
Tomasz Wiktorski

External supervisor:
Jarle Berge

Title of Master's thesis:
Business Management Execution On Data Streams

Credits: 30 ECTS

Keywords: Data Streams • Stream Processing Apache Spark • Apache Storm Apache Kafka • Apache Flink • Corporater	Number of pages: 91 + supplemental material/other: - Code included as link in Appenix Stavanger, June 15 2019

Business Management Execution On Data Streams

Author:
Nicolai Vikanes Stensland

Supervisors:
Tomasz Wiktorski (UIS)
Jarle Berge (Corporater)

June 14, 2019

"Hard work beats talent when talent fails to work hard"

Kevin Durant

Abstract

The world of business management is largely populated with data warehouses as a single source of truth. However, in recent years a shift towards the data origin known as data streams have arisen. Corporater, which is one of the leading companies in delivering business management solutions, acknowledges this trend and wants to investigate their possibilities in supporting data streams. The data streams are most valuable when they are processed and analyzed by a stream processor. This is because the singular events by them selves are less meaningful than a collection of manipulated events. Hence, part of the investigation includes finding the most suitable stream processor for Corporater. Further, it also must be proven that it is possible for stream processors to integrate with Corporater's systems and values.

Through this thesis, we aim to provide this investigation for Corporater. We achieve this by evaluating and developing a prototype which abides Corporater's requirements and environment. Furthermore, the evaluation provides a general overview of the leading architectures and processors for data streams from a business perspective, which conclusively recommends a solution for Corporater. Additionally, the generality of this evaluation allows for beneficial value of other businesses in a similar situation.

Based on the recommendation from the evaluation, we create the prototype. This prototype is largely focused around one of Corporater's core concepts. This is the 'Business-In-Control' concept, which involves having the business experts manipulate and visualize data without the need of a developer. Thus, we implement the prototype in a generalized way that enables configuration from a different environment. This generality enables the prototype to be compatible with any type of GUI or API that are able to send configuration to the prototype. We test the prototype by performing an experiment. The experiment proves to be successful in creating an end-to-end connection with the data streams and Corporater's software. Hence, also proving that it is possible for today's stream processors to integrate with Corporater's system and values.

Acknowledgements

I would like to thank Corporater for giving me the opportunity to work with them in a provident field of research. I am also most grateful for having Jarle Berge as my external supervisor, he has helped me with his expertise and enthusiasm every step of the way. Further, I would like to thank my internal supervisor Tomasz Wiktorski for providing feedback, possible pitfalls and opinions throughout the semester.

I would also like to express my gratitude towards my girlfriend, friends, family and fellow students for keeping up my courage and motivation throughout this challenging semester.

Contents

Chapter 1

Introduction

1.1 Motivation

Businesses around the world have huge amounts of data constantly being stored and analyzed in large databases. All this data is of huge value for the company to see progress and discover issues. In many of these cases, analyzing all this data takes a considerable amount of time. Consequently, some businesses have no time to act on the analyzed data before its too late. Data streams and stream processing can in these cases become the solution.

Data streams can be as an unbounded flow of information. This information can vary in both size and number, where businesses such as Amazon, Google and LinkedIn can use this information for acquiring insight from a certain environment or system. There exist two different types of streams (i) Event streams, where the stream can be separated into separate events, such as a user clicking on a web page, or (ii) Continuous stream, where it is not possible to separate the stream, such as a continuous flow of sensor readings. Furthermore, different systems exists for reading these streams efficiently. These systems are known as 'Stream Processors' and have become a hot topic in recent years. Stream Processors are able to provide quick results for the user to act upon. Usual calculations can be filtering, aggregation and joining. Additionally, some of the stream processors can also provide machine learning libraries such as forecasting or predictions on the data streams. In current stream processing technologies, there is a lot of changes and improvements constantly happening. The increasing need of quick results in the business world, will require numerous businesses to integrate stream processing into their systems.

The company Corporater wishes to explore this topic and integrate compatibility for their system in the future. Corporater delivers a Business Management Platform (BMP) where business experts can build their own business objects that perform complex calculations and visualizations. The objects are based on the 'Business-In-Control' concept. That is, the business experts can perform config-

urations without involving typical programmers (the concept might also be called 'no-code' in some businesses). The business experts assume to have an easy way of setting up the calculation's data sources, and even support precalculations and aggregations. Until now they have only supported batching data where they are potentially updated once a day, week or month, but are now realizing that data streams are part of the new future of business management.

Their platform is intended for medium and large companies, that take use of these complex calculations and visualizations to analyze their data towards improving performance or progress to a goal. Examples of such companies are 'Johnson & Johnson', 'Airbus' and 'Aker BP'. These companies are all customers of Corporater, which makes them all a possible use case for our system. With a large spectrum of use cases, the platform needs to cooperate with many different technologies, where forms and sizes of the data are diverse. Thus this ability of compatibility and flexibility is also desired for real time data, that are increasingly demanded by customers.

1.2 Problem Definition

Typical large enterprise businesses have competent business experts that analyze data and perform management actions when necessary. In many of these cases, new data is only accumulated once per day, week or month. However, new data can also be accumulated in shorter intervals such as seconds, minutes and hours through data streaming. This would enable business experts to make real-time decisions for the business. For example, being able to know about a systems failure at the moment it happens rather than getting a report about it the next week. A large enterprise will typically have real-time production systems to handle it. However, the data events do not aggregate to the business management systems until the week or month is complete. The business would be able order a new module on the same day and save at least a week of production silence. With this example it is easy to see that the value of knowledge is higher the earlier it is received, and the businesses could largely increase profit by incorporating data streams into their systems. Although, these types of data are often not available for the business expert since correct processing of data streams are complicated and technical.

On the other side of the spectrum is the developers, they have extensive technical background that allows them to use data streams with ease. However, in many cases they do not know which data segments can be of value for the business, and are essentially possessing this information without knowing its business value. This leads back to the same problem as in the previous paragraph, which states that the business is not able to draw value from the data streams.

From these two different view points, it is clear that both the business expert and the developer can benefit of each other's knowledge. Thus, from a developers perspective this thesis will focus on solving this issue for Corporater through finding and evaluating the best solutions and further recommend the most beneficial ap-

proach. Additionally, based on the recommendation, a prototype is developed to serve as a proof of concept. With this prototype in place, it would enable business experts to utilize data streams through an understandable and generalized configuration that emphasizes the possibilities of streaming data through stream processing.

In later chapters the prototype will be introduced. However, the values of a business are not necessarily the same everywhere and Corporater is no exception. To figure out Corporater's needs and values, we create a survey. This survey and its resulting requirements will be introduced for the rest of this section.

1.2.1 Questionnaire

Requirements of the system are important to be identified in the early stages of development, such that the system can be pointed in the right direction from the start. To do this, a survey was made for a selected group of Corporater employees to participate in. The group contained 5 employees each in their respected fields to ensure a diverse knowledge base for the questions. The questions were as follows:

1. What are the basic achievements and goals of the system?
2. What kind of roles will be involved in...
   (a) The administration and configuration of the system?
   (b) The end usage and consumers of the system?
3. Will the roles typically be clearly separated regarding to organizational structure?
4. What are the concerns about this type of system?
5. Will the system need to integrate with any other type of software?
6. What operating system will the system be used in?
7. What data streams are the system supposed to work with?
8. The simplicity of installation?
9. What types of filtering or calculations must the system be able to do?

The survey was answered in one meeting, where each member of the group could speak their opinions and wishes to such a system. After the survey, their answers were narrowed down to specific requirements defined in the next part.

1.2.2 Requirements

In this part the requirements of the system will be defined. These requirements are based upon the survey made previously.

The system is intended to function as a micro-service compatible with Corporater's BMP and will make this platform become closer to OI (Operational Intelligence) data sources. It will either open up the platform to dock existing OI solutions into it, or provide a lightweight framework for connecting data sources directly. Furthermore, Business experts are expected to utilize these data sources without relying on a developer, neither on implementation or changes. Corporater's primary goal is to provide an environment for business execution that relinquishes the need for a programmer. However, the business experts are trained in the system and will have access to extensive documentation through Corporater's Academy.

Corporater values flexibility of its platform and are already supporting a large variety of customers in many different fields. This means that the system needs to support as many types of data streams as possible such that all the customers can benefit of its service. Moreover, in many cases Corporater would prefer not to require a customer to purchase another company's license when committing to BMP. This is to avoid needing a third-party license when customers choose Corporater. Software such as Microsoft Azure Stream Analytics [1] and Amazon Kinesis [2] are examples of such systems that would require external licensing.

As of now, Corporater supports Linux and Windows, and this system should do the same. However, the platform's cloud strategy makes this less relevant. Furthermore, the installation is done through a plain and easy wizard that has its own control center to setup general configurations. When BMP is initialized, a business development tool called Configuration Studio is utilized to configure Corporater objects. Moreover, it is intended that the system administrator takes care of the installation and configuration. Further, the business users will utilize Configuration Studio to configure their version of BMP. There is no developer or programming needed in any of these stages. Thus, this system should function the same way, where some of the configuration is done in the control center and the rest in Configuration Studio.

The BMP is not meant to be a critical system that needs on time alerts and updates. This means that the users can tolerate some delay, and not so frequent updates. Corporater's requirement is at least hourly updates, but the option for more frequent updates is not discouraged. On the other hand, the prototype should be able to handle high throughput which would otherwise limit the use cases for an eventual integration.

In most cases batch data will be paired together with data streams. Thus, using the same API for both batch and streaming would be most beneficial. In fact, the beneficial value for Corporater of this feature is higher than other disadvantages that can come with it. Moreover, the users should also be able to do advanced aggregations, even though the majority of calculations are simple. Additionally, it is also desirable to have as many calculations as possible.

1.3 User Stories

A user story describes a feature of the system from the users perspective. These stories are short and straight to the point of what a user should be able to do with the system. By utilizing this concept we are able to describe certain use cases for our system in a way that will highlight the systems features. There are several possible use cases for such a system. Thus, this section will introduce different user stories that represent the end goal of the system. We will utilize a call center as a hypothetical scenario in which a user of our system is accumulating data from.

- As a business expert I would like to have access to current events from the call center without the aid of a programmer.
- As a business expert I would like to visualize current status of the call center in any chart of my choosing.
- As a business expert I would like to aggregate most recent data from the call center.
- As a business expert I would like to select information of my importance from the call center.
- As a business expert I would like to setup a new call center connection without the need of a programmer.
- As a business expert I would like to aggregate and visualize the last 3 hours of my performance metrics.

With these user stories in place, it should be extended clarity of the systems goals and possibilities for the users.

1.4 Challenges

Most similar management systems like Corporater's BMP rely on data warehouses to provide data to their system. However, Corporater sees a trend where the old 'single source of truth' are starting to shift towards the data origin itself or where the data event happened. This creates a new domain where not much work has been done before.

In the stream processing world, there exists a large amount of stream processing software. All of these have different advantages and limitations that are crucial to know about when appointing a stream processor to integrate with. There is a huge challenge in navigating though these to find the most beneficial stream processor for Corporater's scenario.

Because of the large customer base of the BMP, there are several different use cases which our system must abide for. Keeping the system generalized and flexible to

this degree is a challenging effort to complete.

1.5 Contributions

In this thesis we provide an evaluation of the leading architectures and processors within data streaming from a business perspective. This perspective enables other evaluation metrics to be highlighted in a way few other evaluations do. It is mostly focused around Corporater's scenario. However, because of its broad overview of the architectures and processors differences, it can also be applicable for other businesses that wishes to implement OI into their systems. Based on this evaluation, we also provide a generalized prototype that can be applied under any type of GUI or API that is able to send configuration. This prototype brings us one step closer to enabling business experts to utilize data streams without the need for a developer. Furthermore, both the evaluation and prototype aids Corporater's research in providing data stream support for their customers.

1.6 Outline

Chapter 2: Background Presents the technical background required for this thesis, including an introduction to related measurements of relevant stream processors performance.

Chapter 3: Solution Approach Introduces different stream processor setups utilized in the evaluation, and an overview of the components of the developed prototype.

Chapter 4: Evaluation Provides an evaluation of relevant architectures and stream processors where we conclusively present a recommendation for Corporater's scenario.

Chapter 5: Proof of Concept Utilizing the previous recommendation, we present and analyze a prototype based on Corporater's values.

Chapter 6: Conclusion & Further Directions Concludes the thesis and suggests further directions.

Chapter 2

Background

Before introducing an evaluation of different data streams and constructing a prototype. We will present relevant background material of stream processing concepts, architectures and software. Additionally, some business knowledge is also required, such as the difference between Operational Intelligence (OI) and Business Intelligence (BI). Throughout this chapter, these subjects will be introduced one by one, starting with what different architectures stream processors fit into. Furthermore, the information presented here is utilized across all further chapters.

2.1 Architectures

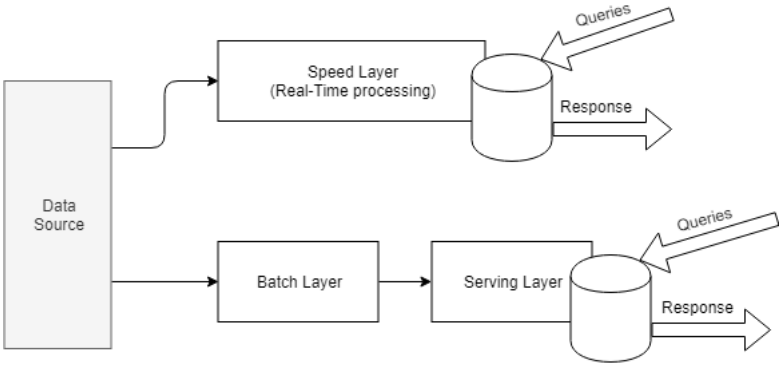
There exists several different architectures for streaming systems. These architectures can be thought of as a template for how data streaming should be implemented. By utilizing these architectures enables organization and simplification of complex systems, which aids to an easier workflow and quick deployments of the overall system. In this section some of these architectures will be introduced. However, the main focus will be on the Lambda and Kappa architectures, which will be utilized in further chapters.

2.1.1 Lambda Architecture

The Lambda Architecture, is a generalized structure of how data can be processed in different layers. It was created by Nathan Marz that later produced a book with this architecture called Big Data [3]. Moreover, it is an architecture that describes the relationship between batch and stream-processing methods [3]. The core idea behind it, is to separate the different data processors into different layers: Batch, Serving and Speed layer. Both the Batch and Speed layer are processing the same data. However, the Speed layer delivers quicker and less accurate results, whereas the Batch layer uses more time and are more accurate. Furthermore, this archi-

tecture is the description of how these layers interact with each other, which is illustrated in Figure 2.1.

(a) Separate Lambda Architecture.



(b) Combined Lambda Architecture.

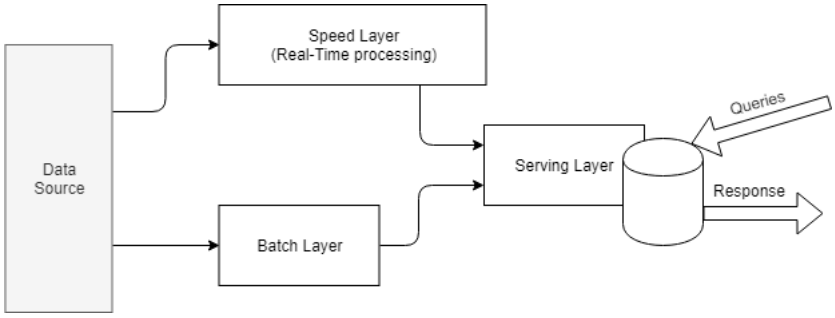


Figure 2.1: Lambda Architectures.

From Figure 2.1a it is possible to see that the Speed layer works on its own to handle queries and responses without going through the Serving layer, while the Batch layer works with the Serving layer. This is not necessarily the case in every architecture. For instance the Serving layer can also handle queries and responses from the Speed layer, thus combining the other two layers in one common serving interface, which is shown in Figure 2.1b. Choosing between one or the other can be different for every implementation, and it really depends on what fits best in each use case.

2.1.2 Kappa Architecture

The Kappa Architecture is a simplification of the Lambda Architecture. Essentially, it can though of as the Lambda Architecture without the Batching layer. It was first introduced by Jay Kreps in 2014 [4] who meant it was much simpler to work with this type of architecture. In most cases, the Lambda Architecture needs two code bases, one for streaming and one for batching, whereas the Kappa Architecture only needs one, which handles both batching and streaming. This simplifies and

reduces the code and makes it easier to do changes, and knowing their effects instantly [5].

In other chapters, a deeper discussion of the two will be conducted, where more of their differences are highlighted in order to draw a recommendation of architecture to the prototype.

2.1.3 Others

The architectures explained previously in this section are the most popular architectures to implement stream processing with. However, there also exists other less popular architectures. One of these architectures is the Butterfly architecture. This architecture aims to provide a unified data store that supports all analytical workloads. Compared to the others this architecture does not have any layers, which might be a benefit in some cases. However, it is relatively new and few existing applications [6].

Another architecture that is described in Nathan Marz's book [3] is the Incremental architecture. This architecture is considered to be a description of a system with no architecture, where all of the system features are incrementally appended to one code base. This architectural approach, could increase the complexity of the system and cause a unnecessarily complicated development and maintenance job.

2.2 Stream Processing

Stream Processing is a useful tool for processing multiple streams of data. Users can expect quick results from the processor, with up to date information of the current state of the system. It is popularized by Apache Storm, that is similar to Hadoop but can give results faster [7]. Still there exists many other contenders in this topic, with different approaches to the problem.

There is vast amounts of use cases for Stream Processing, and with the development of IoT (Internet of Things) it only becomes larger and more important. It is most useful in cases where detection of a problem is possible and an answer must be given in a short amount of time. Moreover, it plays a key role in data-driven organizations. Some applications for stream processing are listed below [8].

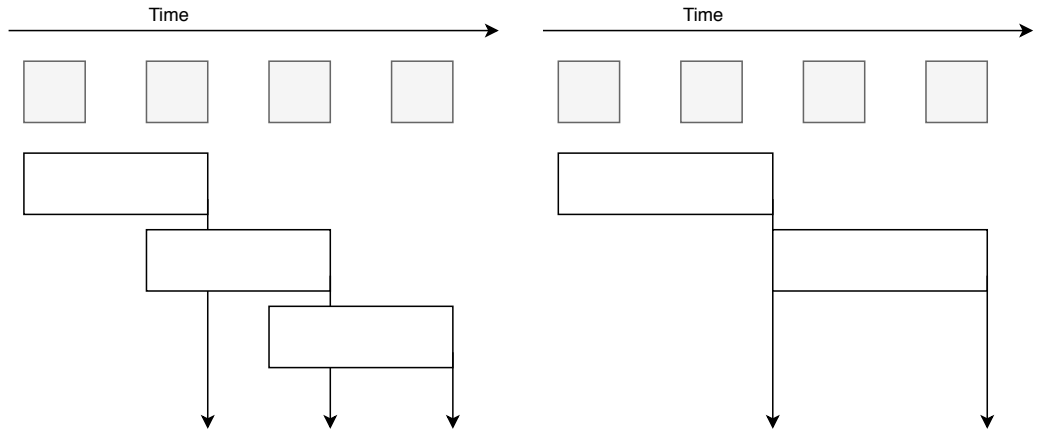
- Health informatics
- Astronomy
- Telecommunications
- Electric grids and energy
- Geography
- Transportation

Stream Processing introduces some new concepts that play a key role in analysing incoming data. The concepts are: Windowing and Joins, it is correct that Joins already exist in batch processing. However certain intricacies requires modifications

to the well known concept, which makes it a new term within streaming. Both Windowing and Joins will be further introduced in the following parts.

2.2.1 Windowing

A window is an input size that defines the number of events that can be stored in working memory. There exists two different types of Windows, Sliding Windows and Batch Windows, where each has a special way of storing the events.



(a) Sliding Windows. (b) Batch Windows.

Figure 2.2: Different types of Windows.

When an event happens with Sliding Windows, the new event will replace the oldest event in the window, and send an update with calculations on the new window to the clients. This means that for every event happening, a new message will be sent to the clients. On the other hand, when an event happens with Batch Windows, the new event will be stored in the window until it is filled up. When the window is filled up, calculations on the window will be updated to the clients. This reduces the number of updates sent compared to the Sliding Windows, and could be better to implement in cases where network resources are limited.

Number of events are not the only size the window can be based upon, time can also function as a window size [9]. This enables users to get data from the stream within a recent time frame such as a minute or 15 minutes. Furthermore, both of the two window types discussed above can be used for time, where it then will be a 'Sliding Time Window' for sliding window, and 'Batch Time Window' for Batch Window.

Using windowing opens up more opportunities for analyzing data streams, such as stock market technical analysis, prediction of next value and multiple moving averages or medians (e.g. 1 minute, 5 minutes and 15 minutes). For the rest of this

thesis we will only utilize batch windows. Thus, when 'window' is mentioned it will be considered as a batch window.

2.2.2 Joins

Joins functions similarly as in SQL, where two tables are joined together based on a common key. Streams can be considered as infinite long tables, but the time it would take to join something infinite is infinite, which is a problem. Windowing can help with this problem by looking only at a part of the stream. It is then possible to join the streams piece by piece. To do this, at least one stream must implement a window that can compare the incoming values from the other stream. With windows on each stream the whole window can be joined at once. These two scenarios are illustrated in Figure 2.3.

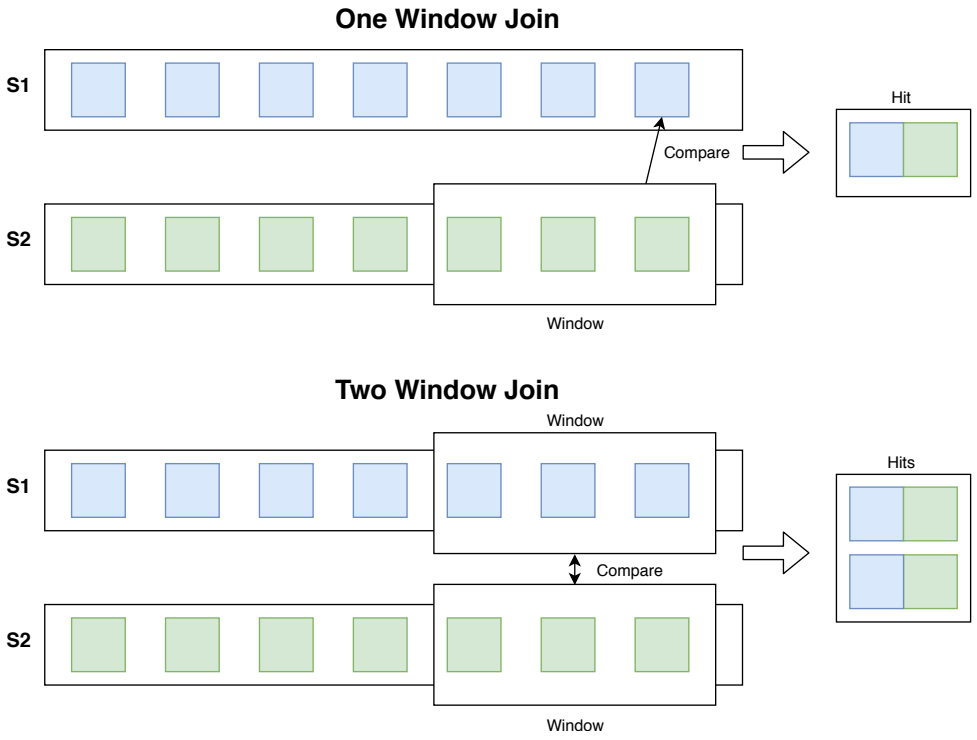


Figure 2.3: Different ways of Joining two streams.

With Joins it is possible to combine two different streams and look at their differences. Usually this is done by converting the window to a table and joining it with the same key. Database tables can also be joined together with the stream. This can be quite useful in occasions where ID's, provided through the stream, can be complemented with other information from a table lookup in the database. Another use case for this, can be for incoming production data where the production status is compared to yesterday's production.

2.3 Streaming SQL

Streaming SQL is a term within Complex Event Processing (CEP), that enables users to do complex calculations on incoming events. As the name deducts, Streaming SQL makes it possible to do SQL queries on streams, and get quick results back with the most recent events.

Regular streaming processing requires a lot of code to get the desired output of the stream. While with Streaming SQL, enables actual SQL queries to be executed on the stream. This reduced the required code length drastically. However, unlike SQL, Streaming SQL has no standard syntax. This means that different languages exists for Streaming SQL. Two examples of such languages are Siddhi Streaming SQL and Kafka KSQL, that are SQL based but for example the queries have some differences one must be aware of before use:

Siddhi	KSQL
Select bid, from $BoilerStream[t > 350]$	Select bid, from $BoilerStream$ Where $t > 350$

In this example there exists a boiler that has a sensor for measuring its temperature. The sensor will function as a stream of temperature measurements, where we want to detect temperatures greater than 350°C. In both languages the system is asked to 'select events from BoilerStream with property t greater than 350'. From the table it can be shown that the 'Where' statement in KSQL is excluded in Siddhi. This is because Siddhi is more compact and have the 'Where' statement within '$BoilerStream[t > 350]$'. The other operators are written in the same way for both of the two languages.

2.4 Streaming Technologies

A wide variety of technologies that supports streaming are available and can be used for this project. Some of these can also be used together to form a stronger service. Relevant technologies are introduced in this section, these will be later evaluated according to Corporater's requirements.

2.4.1 Apache Kafka

Apache Kafka[10] was originally used as an interface between data sources and data processors. Its key components at that time was being able to connect to any type of source, persisting events in queues, and sending them to any type of system.

These components are still quite important to this day, yet additional features are implemented together with this structure. This enables Kafka to do simple stream

processing through the library Kafka Streams, and also support Streaming SQL with their own SQL language called Kafka SQL (KSQL).

Kafka uses Zookeeper as its coordinator that manages coordination and failure recovery of the brokers. The brokers are nodes that handles topics, stores events, and does the processing. This is illustrated in Figure 2.4.

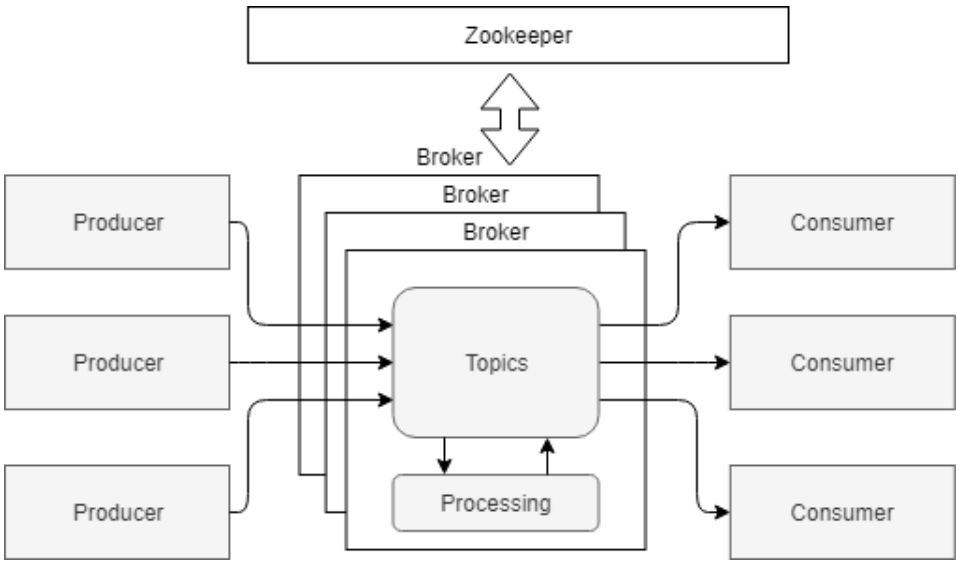


Figure 2.4: Kafka Architecture.

The figure shows that there can be multiple brokers and topics, where the topics can be across multiple brokers. A topic can be considered as a hub, where some systems write to the hub and others listen. This enables Kafka to streamline the processing of events, such that a producer can write to one topic, process it through Kafka Streams, write to another topic and output the topic through a consumer. Intuitively, Producers write events into the Kafka cluster while the Consumers read from the cluster.

Kafka can guarantee 'Exactly-once Semantics', which is the guarantee that an event is processed exactly once. On the other hand, this requires a lot of management from the system which slows the process down drastically. However, since this is a relatively new addition to Kafka it will hopefully be improved in the future. Otherwise, Kafka is quite flexible by allowing other semantics in to the picture such as 'At most once Semantics', which guarantees that an event is processed at most once, and 'At least once Semantics', which guarantees that an event is processed at least once.

2.4.2 Apache Storm

Apache Storm[11] is a distributed real time data analytics system for processing data streams. The system is quite flexible and can work in many different situ-

ations. It is fault-tolerant, horizontally scalable, and has one of the highest data ingestion rates. Similarly to Kafka, Storm uses Zookeeper to keep track of their nodes, which is why Kafka and Storm can sometimes work well together with a common Zookeeper instance. Besides this, there are some extra components in Storm as illustrated in Figure 2.5.

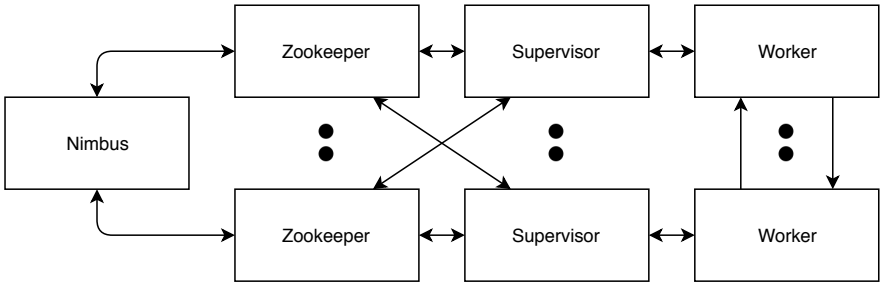


Figure 2.5: Storm Architecture.

This figure shows that there can be multiple instances of Zookeeper, Supervisor and Worker, in which all of them communicate hierarchically with each other. Together they form the Storm cluster, where stream data can be sent through Spouts and Bolts, which are illustrated in Figure 2.6.

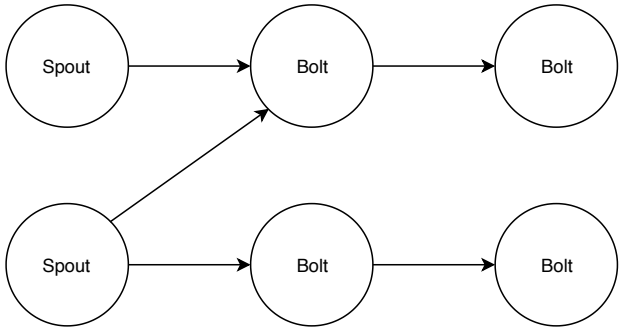


Figure 2.6: Spouts and Bolts in Storm.

Spouts are the stream input, similar to producers in Kafka, while Bolts are the stream processing engines. The Bolt's functionality is similar to how MapReduce is in Hadoop, where the Bolt has the ability to be both Map or Reduce. Moreover, the workload of the Spouts and Bolts are distributed across all the workers in the architecture to ensure maximum performance. On the other hand, it cannot guarantee 'Exactly-once Semantics' without incorporating the high-level API called Trident, which is based on mini-batching. Although, Storm core comes with 'At least once Semantics' built in, which is more than enough in most cases.

Storm is one of the most used Stream processing systems, and has the same stable presence here as Hadoop has for batch processing. Mostly this is because of its low latency and immense community that few others can match.

2.4.3 Apache Flink

Apache Flink[12] is a processing engine for stateful computations over unbounded and bounded data. It was originally called 'Stratosphere' before it became part of the Apache Software Foundation, and it was made to support both batching and streaming in the same infrastructure. The previous mentioned software is more lightweight compared to Flink that enables more advance computations with a simple framework. Its architecture is illustrated in Figure 2.7.

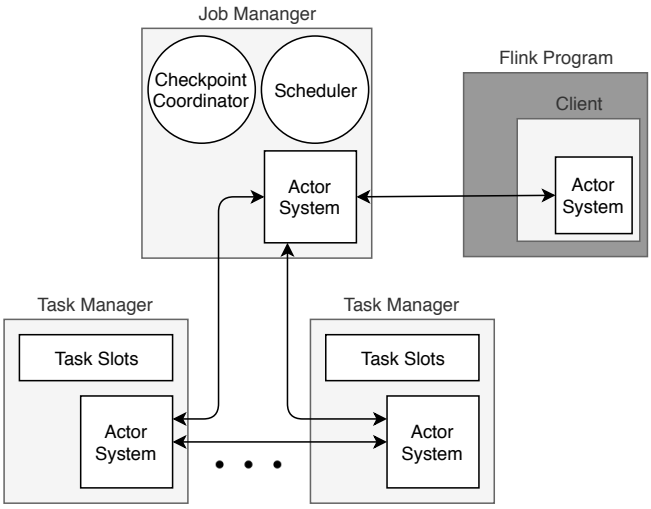


Figure 2.7: Flink Architecture.

The architecture of Flink is completely different in terms of naming and hierarchy compared to Storm and Kafka, it is simpler and contains less components in the overall system. However, the general architecture is similar where there are worker nodes (Task Managers) that compute results, and a Coordinator (Job Manager) that schedules and coordinates jobs to the worker nodes.

Flink is still a bit of a newcomer in the streaming world, but the features that Flink has to offer is sometimes more than its competitors, also it is widely accepted by large companies like Uber and Alibaba [13]. Lastly, Flink can guarantee 'Exactly-once Semantics' together with the underlying categories 'At least once Semantics' and 'At most once Semantics', which is one of the features that Flink offers.

2.4.4 Apache Spark

The previous software thus far are native stream processors. Apache Spark[14] on the other hand is a native batch processor. Spark originated as a successor for Hadoop, that introduced advance features like Machine Learning, SQL support and streaming. In this context, the streaming feature will be the focal point, where the other features are beneficial but not the most important part.

Spark streaming is a library that enables Spark to support streaming. To emulate streaming characteristics it uses micro-batching on the stream, where each batch can contain multiple events or values. Because of its core structure of batching, it cannot compete with the pure streaming implementations that has much lower latency. This could mean that Spark is excluded for evaluation on lower latency required systems. However, data in Spark is processed reliably and can guarantee 'Exactly-once Semantics', together with the other underlying categories. With this and the other features it is still a good contender in streaming systems.

Another library that provides streaming is 'Spark Structured Streaming', which is part of the Spark SQL library. Before Spark 2.x this library did not support streaming, but since then, this library is a well know library within streaming. Moreover, Structured Streaming is an attempt on unifying batching and streaming, with only a few differences in method calls and syntax differentiating them. In addition, this library provides a simpler way of working with streaming compared to its predecessor, where Spark is trying to remove unnecessary configuration and fine tuning from streaming, such as batch size, which now Spark can take care of. It also allows to write SQL-like syntax on the streams, which can drastically reduce code length and development time.

The architecture of Spark is quite similar to Flink, where it is a simple structure of worker nodes (Executors) and a Coordinator (Cluster Manager) that works together in the system. This structure is illustrated in Figure 2.8.

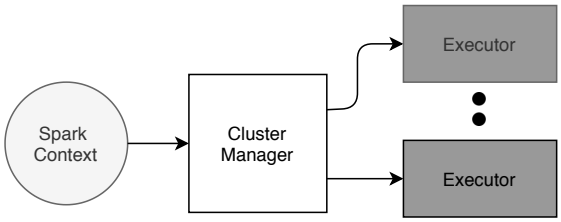


Figure 2.8: Spark Architecture.

Even though the architecture and features are similar to Flink, their approach is completely different. This is mostly because of their initial focus where Spark focused on batching and then added support for streaming, while Flink focused on streaming and then added support for batching.

2.4.5 Others

One stream processor that is not included further is Apache Samza [15]. This processor is intended to be put on top of Apache Kafka to do stream processing calculations and aggregations. However, it is beginning to lose traction to the previous stream processors, in which it cannot keep up in some aspects. Thus, because of limited evaluation time of these processors, it was not included.

Another stream processor that is not part of our consideration is Apache Apex [16]. It provides similar features to Spark in many key instances. However, it is a relatively new system which few large companies are utilizing. Additionally, there are few related benchmarks between Apex and other stream processors, which makes it hard to pinpoint where their system are in the realm of stream processors.

There also exists other stream processors such as Microsoft Azure Stream Analytics [1], Amazon Kinesis [2] and Google Cloud Dataflow [17]. However, these are not open source systems and were not considered further based on Corporater's requirements.

2.5 Business Intelligence

Business Intelligence (BI) is an umbrella term for extracting value out of data for the business [18]. This term is used in many different cases, but was first introduced in this form by Howard Dresner of the Gartner Group in 1989 [19]. BI-data is usually stored in 'Data Warehouses', where subscribers can get updates once a day, week or month. Extracting business value out of data can be so many things, for example data mining and text analytics are part of the rising topics within BI. Usually the data warehouses provide databases that are accessible through SQL queries, which is where most of BI-value comes from.

BI is at the core of Corporater, where the BMP provides different tools to extract value. One of the main topics for Corporater within BI is Business Performance Management (BPM), which will be described next.

2.5.1 Business Performance Management

BPM is a management tool that is used to optimize business strategy. Businesses that want to improve or expand use this tool to first, define what and where they want to improve and secondly, track and analyze the business towards these goals. BI on the other hand, provides tools to improve decision making within organizations, but have no means of planning, monitoring, controlling and managing goals and business strategy without BPM [20].

The goals are usually described in Key Performance Indicators (KPI), which are measurable values of the goals. KPI's are used to illustrate how effectively a company is achieving their business strategy. If used correctly KPI's can be essential to BPM and the company using it. This requires reliable reporting of current state of the goals, and often reevaluation of indicators such that they can continue to be realistic and in the direction of the business strategy.

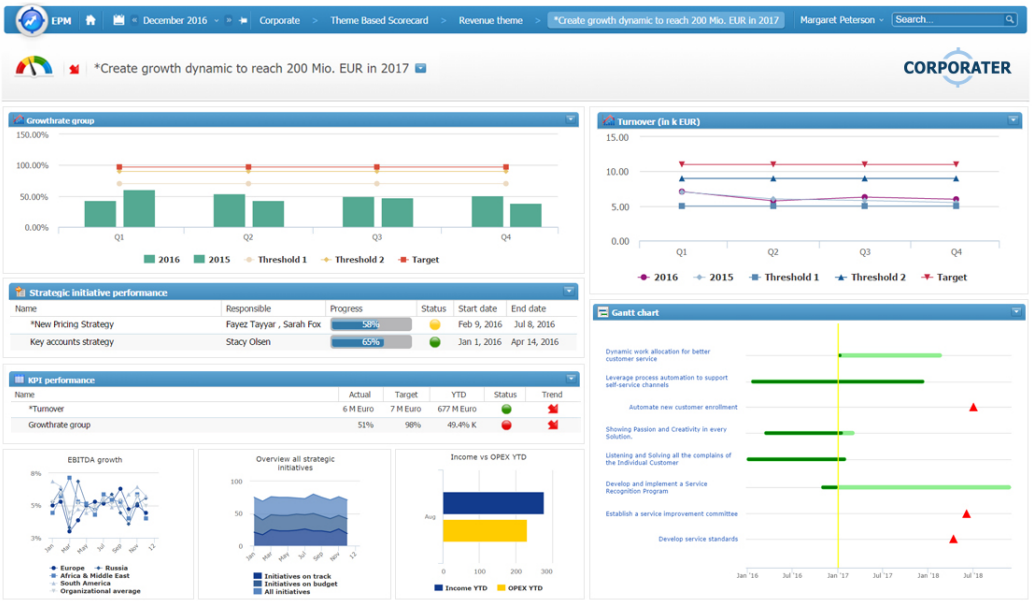


Figure 2.9: KPI Dashboard Example [21].

Figure 2.9 [1] shows an example of some KPI's in a typical dashboard, for instance histograms and graphs are typical. A KPI can be illustrated in many different ways, and choices of formats can be based on what the KPI is meant to represent or sometimes preference.

2.6 Operational Intelligence (OI)

Operational Intelligence (OI) is about giving value to the business through analyzing and handling real-time events. This enables the company to react quickly on problems and opportunities in the daily operation of the business. Typically, the data analysis part is automated, such that only alerts require an action. With this tool, employees can take faster action with more knowledge about the situations arising [22].

Further benefits of OI can give accurate and reliable results from the current state of the system, where otherwise human error can be a factor. OI can also accelerate continuous improvement of the business, where alerts can be adjusted as the business improves. Moreover, OI analytics can also take use of KPI's in the same way as BI but of course in a smaller daily scale [23].

While OI handles short term day-to-day operations, BI handles the analytics for forward planning that OI cant. Both of these intricate parts can complement each other to form a complete picture of the business at hand. Implementing OI into the BMP is something Corporater is reviewing as an expansion to the platform.

[1] This image is approved by Corporater to use for this thesis.

2.7 The Business Management Platform (BMP)

The Business Management Platform (BMP) is a comprehensive management system created by Corporater, which is largely focused around BI and BPM solutions. This system can bind several business disciplines such as risk, strategy, operations, quality, projects, processes, HR, and finance together in one platform. Furthermore, this allows for top-ranking executives to control the entirety of their business, which enables the whole business to follow one strategical direction towards a common goal. An example of their web interface is shown in Figure 2.10.

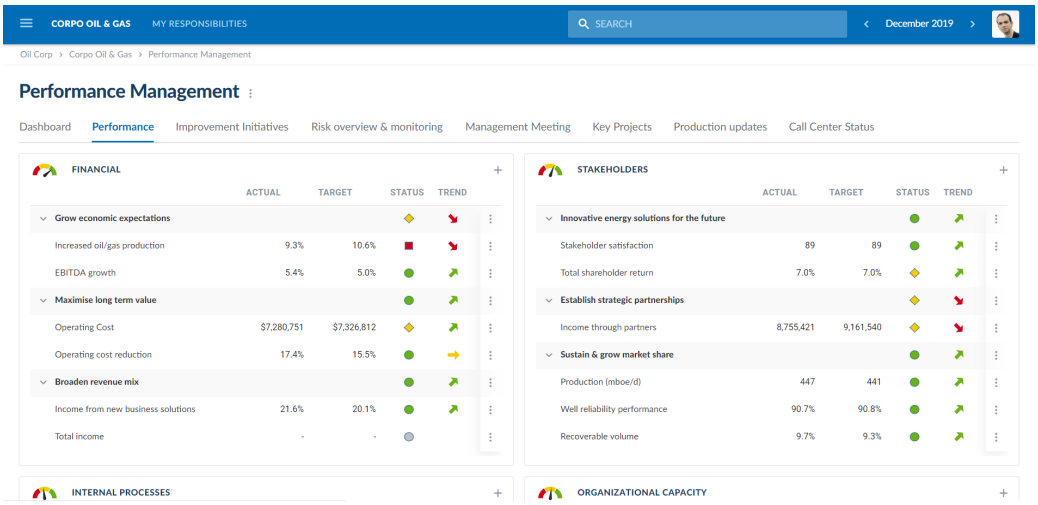


Figure 2.10: BMP web interface example.

From this figure it is possible to see that there are different tabs for each discipline. In our case we are in the performance management tab that can show the status of for example recent financial status. Furthermore, the platforms core concept is called 'Business-In-Control'. This concept is based around keeping the business users in control of the system, which allows for configuration and modification without the need of a programmer. The programmer is relinquished by providing more than 250 business objects that are configurable through a GUI called 'Configuration Studio'. An example of these objects within Configuration Studio is illustrated in Figure 2.11.

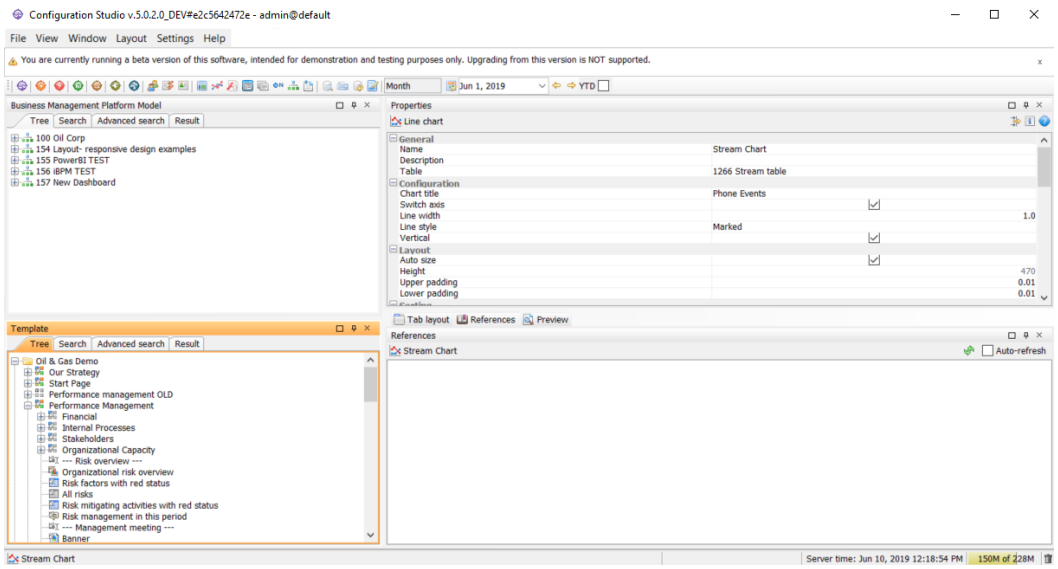


Figure 2.11: The Configuration Studio.

These objects are displayed in the left part of Figure 2.11, which allows the user to store, modify and display important information on the web to maximize BI value. One key feature of their system is that this important information is unified with BPM, which allows easy follow up on current goals and status of the business.

2.8 Related Works

No other papers have investigated this for Corporater. It is unique case where not much work is related to it. Although, others have done their analysis of Stream Processing systems. In this section some of these papers will be presented with their results. Their analysis is one of the factors that is utilized when choosing the best Stream Processor for Corporater.

In the late 2015 there was little to none benchmarks comparing the different stream processors. Yahoo! wanted to know what the best streaming tools are, in order to provide the best service to their internal customers. They designed a simple advertisement application which read JSON events to later filter, transform, join and aggregate linearly through a pipeline. Furthermore, each cluster contained 10 worker nodes, which were 'homogeneously configured, each with two Intel E5530 processors running at $2.4GHz$, with a total of 16 cores (8 physical, 16 hyperthreading) per node. Each node has $24GiB$ of memory, and the machines are all located within the same rack, connected through a gigabit Ethernet switch'. With this setup they tested 'Storm', 'Flink' and 'Spark Streaming' that produced a graph shown in Figure 2.12 [24].

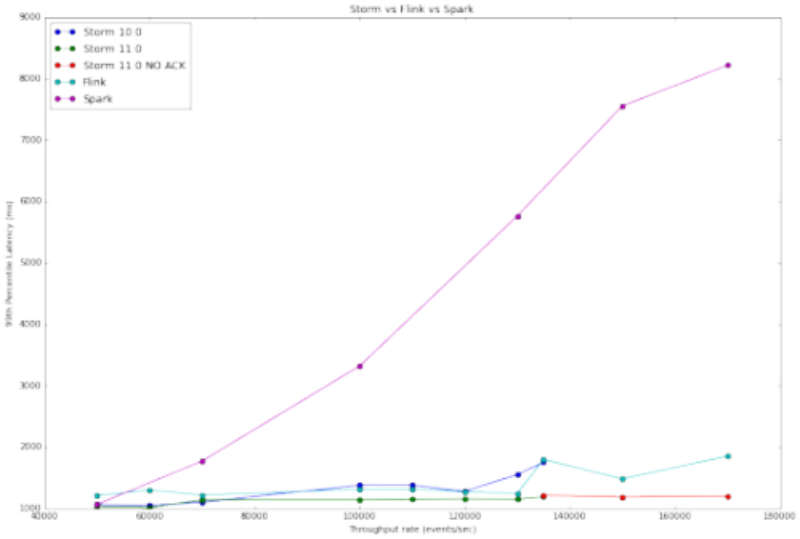


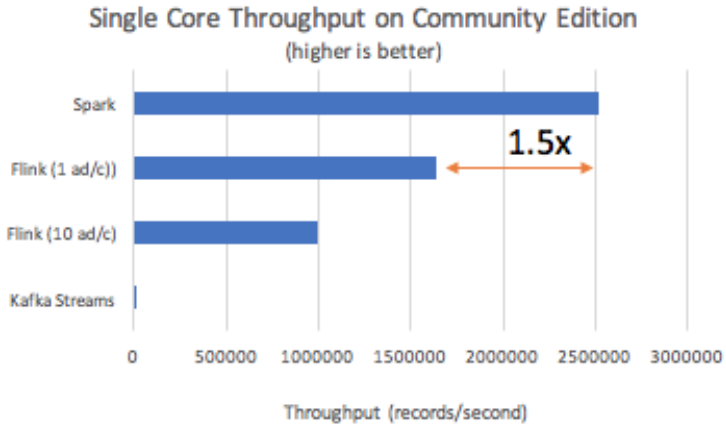
Figure 2.12: Latency and Throughput Benchmarks [24].

This figure shows the latency of the respective processors as the throughput is increased. These results shows that Storm and Flink have similar performance in terms of keeping the latency down when the throughput increases. Spark on the other hand, shows much higher latency's but in turn it is expected to handle much higher throughput's [24] than what is shown in this graph. Kafka is not mentioned in this benchmark, however there are other benchmarks [25, 10] that have achieved latencies around 1-1000ms which is quite similar to Flink's performance.

Yahoo! was one of the first companies to do a large scale comparison between the streaming systems, and it has become a well-known benchmark used in industry to evaluate streaming systems. Although, a multitude of changes have happened since 2015, such as Spark Streaming 2.0, which is why other benchmarks from other sources are necessary. In recent years both Spark and Flink developers have provided their own benchmarks focusing more on throughput rather than latency. Interestingly, their benchmarks shows different results, which are to be introduced further in this section.

Spark 2.x introduced a separate technology based on 'Datasets/DataFrames', called 'Structured Streaming'. Additionally, this version also introduced multiple enhancements to their other libraries. Following this version, new benchmarks where conducted by Spark developers, which tried to use the same experiment as in Yahoo!'s benchmark. These benchmarks are shown in Figure 2.13 [26].

(a) Single core.



(b) Yahoo! environment.

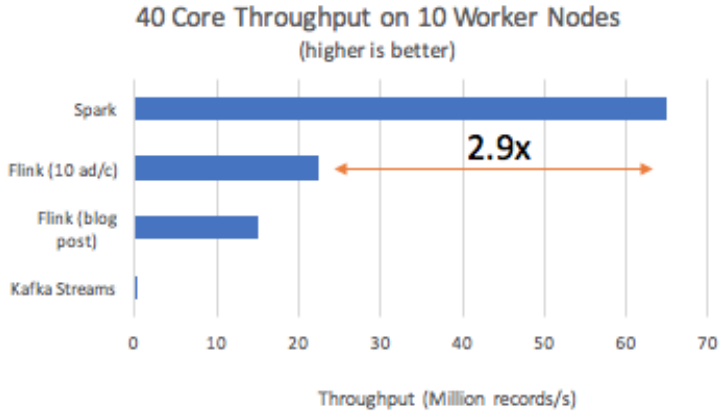


Figure 2.13: Benchmarks from Spark developers [26].

There are two charts in Figure 2.13. The first one shows the comparison of Spark and Flink's throughput on a single core system, where it is possible to see that Spark outperforms Flink by a large margin. Further, in the last figure the replicated Yahoo! environment is shown, where even a larger margin is in Spark's favour. Notice that their version is able to have over 60 million records/second, which is quite impressive. Additionally, 'Kafka' is also shown in this figure, but are not even close to the throughput capabilities of Flink and Spark, which is because of Kafka's larger focus on latency rather than throughput [26].

Following the latter benchmark, developers from Flink became sceptic over Spark's evaluation, thus they produced their own comparison showing much higher throughput's from Flink. As in the previous benchmark, they tried to use the same experiment from Yahoo!'s benchmark, which lead to the chart shown in Figure 2.14 [27].

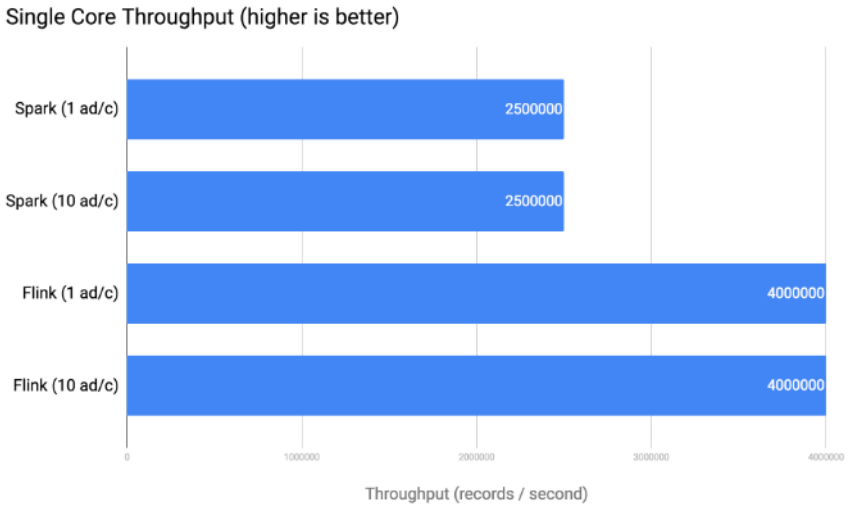
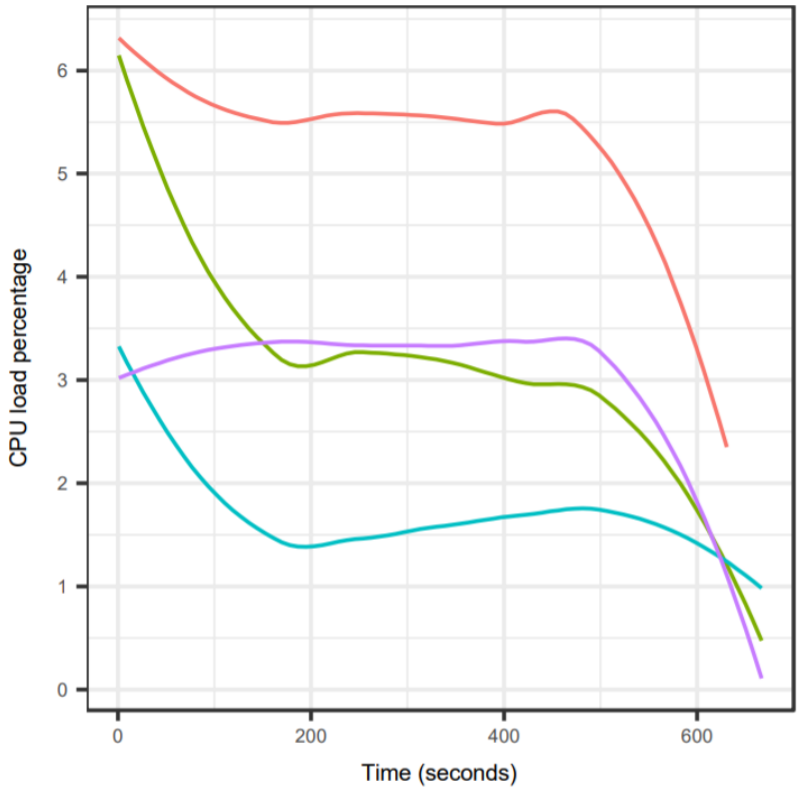


Figure 2.14: Benchmarks from Flink developers [27].

This figure shows the single core throughput of Spark and Flink. It is shown that same numbers from Spark's benchmarks where achieved in Spark, however a significant jump in performance happened in Flink. Although, the throughput of 10 worker nodes is not covered in this benchmark, which allows for speculation whether Flink outperformed also Spark in this case or if it is much more closer race [27]. Furthermore, Storm is not covered in neither of Spark and Flink's measurements, however Storm has done their own benchmark [11] of their system, and are able to achieve over a million tuples processed per second per node, which puts it somewhere between Kafka and Flink in performance.

Throughput and latency are not the only types of benchmarks for stream processing. In recent papers resource consumption has become an important evaluation metric of their analysis. One such paper were conducted last year [25], in which they analyzed Kafka, Flink and Spark regarding the consumption of CPU-resources and memory. Storm was not included in this analysis because of it's low performance on their systems. Furthermore, for their analysis they used 10 nodes where each node contained 16 cores of CPU and 32 GB of memory. The result of this analysis were comprised into a two charts shown in Figure 2.15.

(a) CPU usage comparison.



(b) Memory usage comparison.

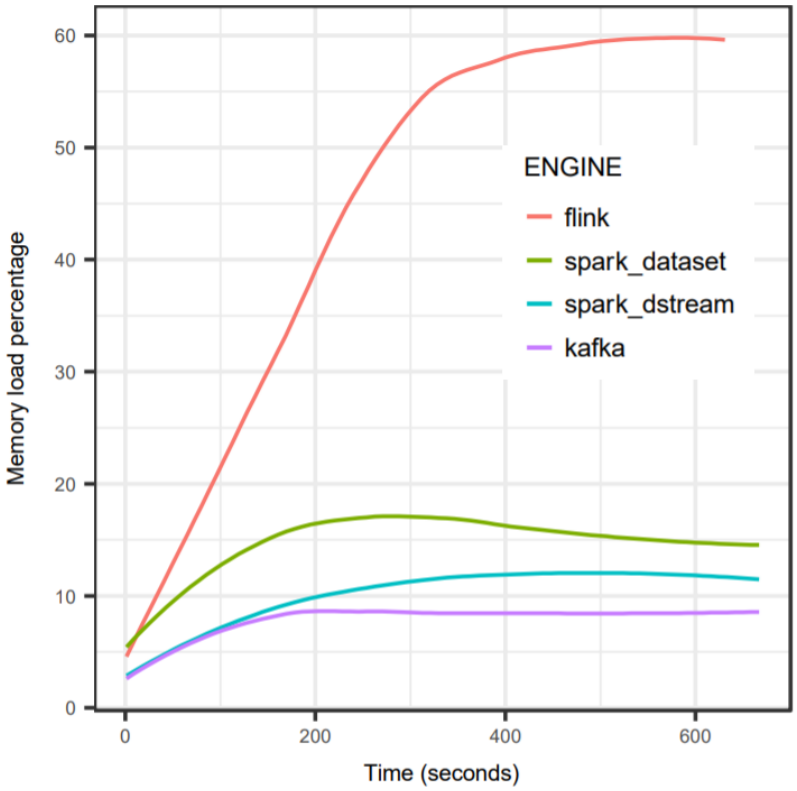


Figure 2.15: Resource comparison [25].

Both of these figures shows the resource consumption over time, where Figure 2.15a shows the CPU consumption and Figure 2.15b shows the memory consumption. A commonality between the two, is that Flink has high values of consumption, whereas the others are quite close to each other. Furthermore, Kafka streams have the lowest memory consumption, while Spark streaming has the lowest CPU consumption. The last system compared in this figure is the Spark Structured Streaming system, which is similar to both Kafka and Spark streaming in consumption, however it's values are a little bit higher. As mentioned before, Storm is not mentioned in this research, however another paper [28] presented the resource consumption of Storm, which provided similar results as Spark. Thus in these figures, Storm can be considered to be in the same region as Spark.

There are still many factors that play in when choosing a stream processor for an environment. These factors will be presented and evaluated together with these benchmarks in Chapter 4. Moreover, Table 2.1 is created here to serve as a final summary for all these related works. Not all numbers in this table are introduced in this section. However, these are found in their respective papers which measure them. Additionally, this table will be the reference point of this work for later chapters to utilize.

Metrics	Kafka	Storm	Spark	Flink
Latency	$<$ 100ms	$\ll$ 100ms	$<$ 1s	$<$ 100ms
Throughput	100-800K rec/sec	$>$ 1M rec/sec	50-60M rec/sec	10-40M rec/sec
Resource Consumption	Low	Med	Med	High

Table 2.1: Summary of related works metrics.

Chapter 3

Solution Approach

Data streams and stream processing is starting to become a hot topic in today's business world, several businesses do not have the knowledge nor the systems to integrate with. Corporater has started their research in this topic to see if there is benefit in integrating such data event characteristics inside their BMP software. However, knowledge is lacking in this field, and assistance is given through this paper by evaluating different stream processors and further developing a prototype. Moreover, this paper serves a recommendation for Corporater's scenario, and creates a foundation for further development and investigation. Additionally, the prototype can be utilized by business experts, which enables them to get an understanding of how beneficial data streams actually can be.

In this chapter, there will be presented four different stream processors and their setup. These stream processor setups will later be used in an evaluation presented in Chapter 4. Thereafter, a prototype is presented that serves as a proof of concept, and will be later evaluated in Chapter 5. Finally, some further directions are given, which serves as an introduction to the later chapters.

3.1 Stream Processors

In the stream processing world, there exists several different stream processors that are able to perform close to real-time computing. Some of these are more popular and more used, while some of them are newly created and are just starting to gain traction. In this section, four of these stream processors have been selected based on the most resent benchmarks [26, 27, 24] and evaluations [13, 29] for stream processors, which suggests them to be one of the best open source stream processors on the market. These processors are called: 'Kafka', 'Storm', 'Flink' and 'Spark', which are all part of the Apache foundation. Each of these processors have been introduced in Chapter 2: Background. However, in this section these are explained more in terms of how they are set up and how this structure is.

A stream processor does also need something to process, and for this thesis a generator will be used as a source for the data stream. For each processor the same generator is used, where the connection happens through a TCP socket. Furthermore, the rest of the section will follow the same order as the processors were mentioned in the previous paragraph.

Kafka requires a low-level implementation where most of the configuration happens through command line. Furthermore, without implementing a script, there are also needed multiple command line interfaces (CLIs) in order to have all the instances running at once. All these instances are illustrated in Figure 3.1.

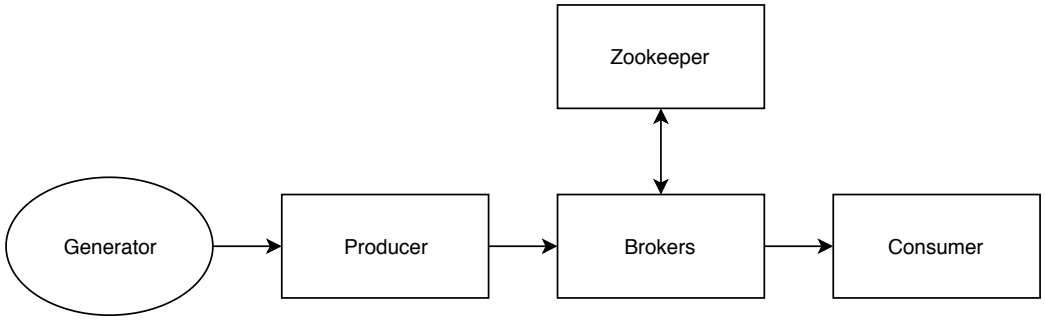
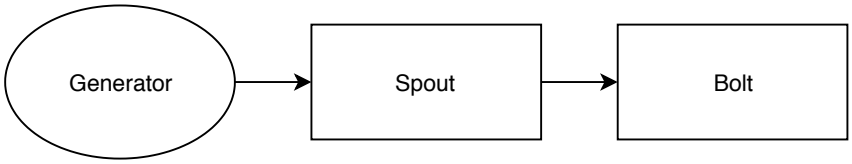


Figure 3.1: Kafka components setup.

From this figure, without counting with the generator, it is possible to see that it requires at least four different CLIs. Additionally, the Producer needs to be implemented in a programming language such as Java or Python, in order to write data into the cluster. Furthermore, setting up this environment without any knowledge prerequisites of this system, requires a high learning curve just for a simple implementation.

A system similar to Kafka is Storm, which also uses Zookeeper in its architecture. Storm is also low-level and most of the configuration happens through the command line. This also means that Storm requires multiple CLIs in order to initiate the whole architecture. However, Storm is a little bit different from Kafka in terms of required setup configuration. It uses a topology feature to determine the structure of the running computations, where Storm distributes the topology out to all the supervisors in the architecture. Thus, both the cluster and topology must be created and initialized just for a simple implementation. Both the needed topology and cluster components are illustrated in Figure 3.2.

(a) Topology Setup.



(b) Background Setup.

Figure 3.2: Storm components setup.

Figure 3.2 is separated into two figures, the first figure shows the topology and the second one shows the necessary cluster components. Furthermore, the topology is created by using a programming language whereas the Background Setup is created using command line inputs. Figure 3.2a shows the most simple topology to create, which requires at least three Java classes to implement. Additionally, this structure is separated from the cluster until it is deployed through a 'StormSubmitter' within the implementation. Figure 3.2b shows the minimum amount of cluster components required to deploy Storm. Each of these three components needs their own CLI, which is in a similar manner as for the Kafka implementation. Another similarity to Kafka, is the high learning curve for a simple setup. However, it is a little bit easier for Storm. This is because of detailed tutorials from Storm and other users [11].

Our next processor is Flink, which is a processor part of the high-level stream processors. That is, it only requires one running CLI. The rest of the configuration is either handled by Flink internals or written in programming code. An illustration of this implementation is shown in Figure 3.3.

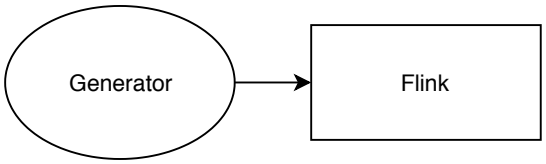


Figure 3.3: Flink component setup.

This figure shows the required components of a Flink implementation. Compared to Kafka and Storm it is quite minimalistic, and not much knowledge of the whole system is required by the user. In fact, the only thing that is required by the user, is to know Flink's API and how to push implementations to the cluster. Additionally, this also limits the number of required Java classes, which simplifies the implementation drastically.

The last processor to consider is Spark, which is a processor that is quite similar to Flink in terms of setup. It only requires one running CLI, where one can either push implementation to the cluster, or write the implementation directly on the cluster through the CLI. This setup is shown in figure 3.4.

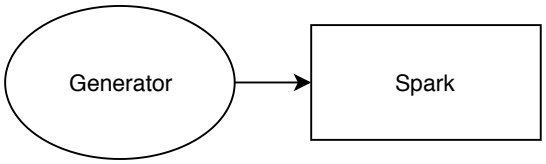


Figure 3.4: Spark component setup.

From the figure, there is not a significant difference between Flink and Spark. However, in reality this is not the case. This will be further discussed later in Chapter 4: Evaluation. Spark requires a low-level of understanding in order to run a simple job, and there is not much configuration needed in order for the cluster to be implemented. It's only requirement is the knowledge of Spark's API and the ability to push implementations to the cluster, which is also similar to Flink.

3.2 Prototype

In this section, there will be described a prototype. The decisions of architecture and stream processor will be introduced as recommendations in Chapter 4: Evaluation. Furthermore, this prototype will also be analyzed in an experiment in Chapter 5: Proof of Concept. Moreover, this prototype serves as a proof of concept for Corporater, in which they can further build upon in the future. The prototype is an end-to-end connection from the stream source to Corporater's BMP software, which allows for BMP users to utilize data streams. In this section, this prototype will be explained, where all of the components required for end-to-end connection are included.

It is not only a stream processor that is chosen for this prototype, but also the architecture. This architecture was also part of the recommendation done in evaluation chapter. Based on this evaluation, the Lambda architecture with the Speed layer being separate to the rest of the structure, was selected to be the most beneficial architecture for Corporater's system. From this architecture it is required to have both a Batch layer and a Speed layer. Corporater already had the Batch layer in place to process data. Thus, only a Speed layer and a Serving layer for the Speed layer were necessary to implement. These layers are illustrated as segments in Figure 3.5.

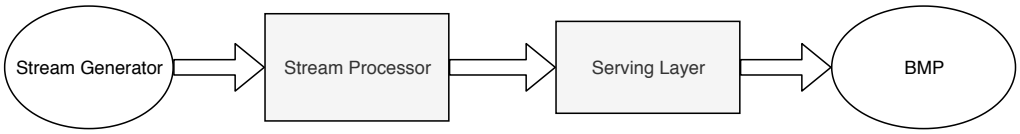


Figure 3.5: A diagram of the different segments of the system.

This figure consists of 4 different components: 'Stream Generator', 'Stream Processor', 'Serving Layer' and 'BMP'. They each have their different responsibilities for the system to function. Except the generator, which can be considered as any random data stream for this chapter. However, it is discussed more in detail in Chapter 5: Proof of Concept. Hence, the segments of importance for this section are the Stream Processor, Serving Layer and BMP. Furthermore, Spark was chosen as the recommended stream processor, and will be our Stream Processor segment. Additionally, the Serving Layer segment can be considered as a database that runs in memory commonly known as 'in-memory', which will be explained more in detail in its respective part of this section.

3.2.1 Stream Processor

The main component of the Stream Processor is Spark which handles all calculations. Communication and configuration is not done in Spark, rather several helper classes are used to accommodate those needs. The dynamics between the different classes are illustrated with a diagram shown in Figure 3.6 which is the architecture of the Stream Processor.

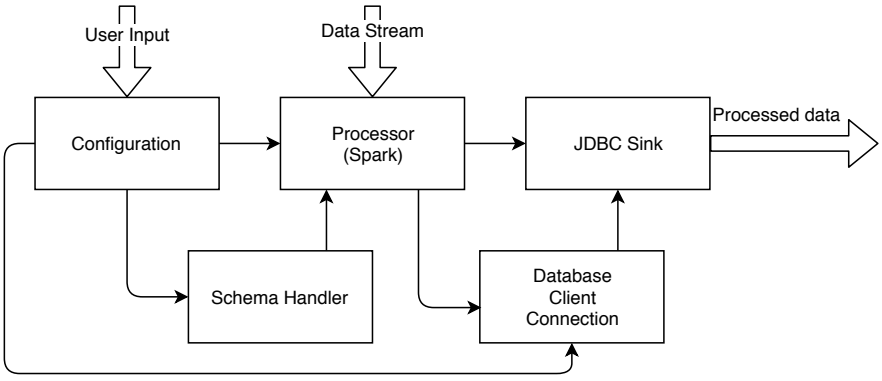


Figure 3.6: Stream Processor Architecture.

Starting from the 'Configuration' module in the figure, this module handles all configuration needed in Spark to process the stream and what calculations to run. In addition, this module handles information about the connection to the 'Serving Layer', and delivers information about the structure of the stream to the 'Schema Handler' module. The Schema Handler takes a structure or sample of the stream

and converts it into a known Spark structure, Spark can then use this structure to interpret incoming events from that stream.

On the other end of Spark there are two modules: the 'Database Client Connection' and the 'JDBC Sink'. The first module is not only used to maintain the connection to the Serving Layer, but also to create a table in which the latter module can write to. Furthermore, as each processed row is created by Spark, the latter module writes each row as they appear into the Serving Layer.

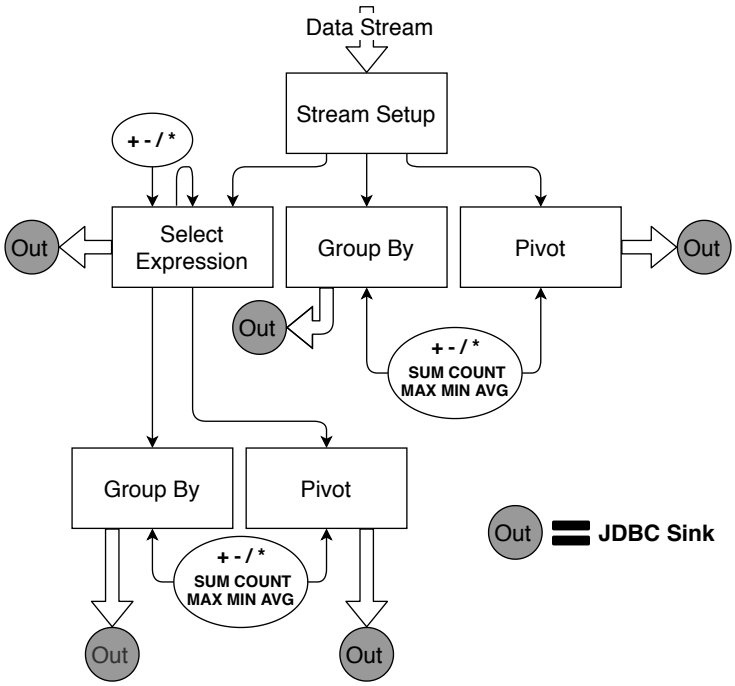


Figure 3.7: Block diagram of the Processor model.

Lastly is the processor module which contains the Spark environment. The 'Structured Streaming' library is the core component in this environment. Still all calculations that are allowed by Spark can be done here. However, the calculations 'Select Expression' and 'Group By' are generalized to be utilized by business users. Additionally, pivot functionality is implemented, which is not something that the Structured Streaming library currently supports. Figure 3.7 shows the methods that can be utilized in the different stages of the module. The figure starts off with the 'Stream Setup' where its parameters for connecting to the stream is provided by the configuration module. From this method, the stream can go through three different methods:

- **Select Expression:** Allows the user to write SQL-like queries where different operators can be incorporated into the expression. Furthermore, multiple expression are possible within the same method call, which enables the user to do diverse calculations on the columns.

- **Group By:** In this method the user can group on different columns. The window size must be part of the grouping to avoid infinite-time computations. After the grouping, the user can then extract more meaning of the events through aggregation.
- **Pivot:** Contains a two step process to achieve pivoting of columns. The user can select the column to pivot and provide a set of all unique inputs in that column, what column to group on and what aggregations to do. This will in turn give a new table with the correct columns in each window batch. However, further improvements are required to use this functionality in a generalized environment.

Additionally to all these methods, it is possible to also do filtering before or after each method. Another interesting fact can be seen in Figure 3.7, the Select Expression is the only method that can be called multiple times and also before the other two methods. This is because there is no grouping of data in this method, and Spark allows such methods to run multiple times.

3.2.2 Serving Layer and BMP Additions

The Serving layer consists of a simple implementation of a H2 database. This database runs in-memory that enables quick reads and writes into its tables. Not much code is required for a runnable implementation and it is quick and easy to setup. Moreover, H2 is a database that the BMP segment supports. This makes it easier for the BMP to reach into the stream without any large implementation at that end. However, some implementation is required.

In BMP, connection to streams have previously been impossible. It is tailored for batched data and their smallest time value is days. Thus, when implementing this feature, finding reusable components and connections is key to avoid further complications. Fortunately, several components were reusable and only an implementation of a stream table was required. Furthermore, this table extends other features, such as charts, which makes it possible to visualize the stream. Visualizing the stream in a chart, enables the user to draw even more information out of the stream. Thus, giving it even more analyzing capabilities.

3.3 Further Directions

This chapter has been separated into two different sections: Stream Processors and Prototype. In the Stream Processor part there were four processors introduced. It was shown how the setup of these processors are, and what was required to implement an instance of them. Later, the Prototype was introduced, which serves as a proof of concept for Corporater. Furthermore, this prototype created an end-to-end connection between the data stream and BMP, which allowed for BMP users to utilize data streams.

In the next chapters, the same separation, as with Stream Processor and Prototype, is separated into two chapters: Chapter 4: Evaluation and Chapter 5: Proof of Concept. In the Evaluation chapter the stream processors are evaluated based on a set of metrics. These metrics are defined in the Evaluation chapter. After each processor is evaluated on these metrics, they are combined to look at the most fitting processor for Corporater's scenario. Additionally, an evaluation is done of the different architectures to also later recommend the most fitting in this scenario. Further in the Proof of Concept chapter, the prototype is first used in an experiment, and later analyzed based on the prototypes capabilities. The experiment will use the same stream generator as in this chapter, only that it will be introduced more properly. After the experiment, some results will be shown that is later analyzed upon certain conditions.

Chapter 4

Evaluation

The BMP software is utilized by several businesses around the world. In order to keep customer satisfaction high and also provide new functionality for them, careful consideration and evaluation of available software is essential. This is also the case when integrating with systems that provide stream processing. Thus in this chapter, we will provide an in-dept evaluation of the leading architectures and stream processors introduced in Chapter 3: Solution Approach. Based on the requirements through the survey in Chapter 1: Introduction, we will find and recommend the most beneficial solution for Corporater.

4.1 Architectures

As introduced in Chapter 2: Background, there are two different architectures to evaluate: Lambda and Kappa. These two architectures are templates or philosophies to go by when developing a system for processing multiple types of data sources. In this section the limitations and advantages of these architectures will be introduced separately. Further to illustrate their differences, these two architectures are brought together in a table that also serves as a summary of this evaluation.

Lambda Architecture

The Lambda architecture is generalized term which assigns different layers with different tasks. These layers are called Batch, Streaming and Serving layer, which all aids developers in simplifying the complexity of the traditional iterative architecture [3]. Furthermore, the Lambda architecture ensures low probability of errors even if the system crashes. This is done through the Batch layer which provides a fault tolerant distributed storage for the historical data. Additionally, this architecture is highly scalable for data processing, which is important in today's systems that need to alter their size upon demand. Another benefit of the Lambda

architecture, is the good balance between speed and reliability, which enables calculations to be as fast as possible without causing the reliability of the system to decrease. Lastly, the ability of reprocessing data, is one of the key challenges of stream processing, which the Lambda architecture handles through running a new job in the batch layer. However, if the same calculations are done in the Speed layer, reprocessing of historical data may not be necessary unless changes to the code have happened [4, 30].

There are also some further disadvantages with this architecture. One of these is the multiple layers needed for a proper implementation. As stated in the previous paragraph, the layers reduce complexity in the system. However, in most cases this will result in multiple code bases, which can sometimes be equally difficult to manage. Additionally, this also makes the system difficult to migrate or reorganize since it would require going through the multiple code bases to get the job done. Another disadvantage of this architecture is the inevitable overhead between layers, which is not required in a layer-less system. Thus, because of this overhead, more data is running through the system creating larger throughput requirements for the same job done in a layer-less system [4, 30].

Kappa Architecture

The Kappa architecture is a simplification of the Lambda architecture, where the batch layer is removed from the system. The biggest advantage of this architecture is that reprocessing is only done when the code changes. This is done through creating a new streaming job which reads the historical data with the new computational changes. Furthermore, Kappa requires less resources in the overall system compared to Lambda that is running the batch layer in parallel. Additionally, only one code base is required for data processing, which lessens the complexity of the system drastically. Lastly, this architecture can be deployed with fixed memory, where for example only the past 30 days are stored rather than all the data [4].

There are also some disadvantages with this architecture. One of the most profound issues is a little bit of the same as its advantage. This is the simplification of the data processing layers into one Speed layer. For instance, when reprocessing the data with recent code changes, it requires having temporarily two times the storage space such that the historical data can catch up with the data stream. Additionally, this reprocessing technique requires that the receiving database support large amounts of writes at once, which is something not all databases supports. Similarly to the Lambda architecture, the Kappa architecture does also have the same overhead between layers, which layer-less systems do not have. Lastly, it is possible that the absence of the batch layer can cause errors during data processing or while updating the database [4, 30].

Summary

Now that the advantages and disadvantages are established of these well developed architectures. They can be highlighted in a table, such that their differences are more clear and intuitive. This table is shown in Table 4.1.

	Lambda	**Kappa**
Code bases	2	1
Reprocessing frequency	All the time	Upon code change
Extra storage	No	Upon reprocessing
Overhead between layers	Yes	Yes
Layers of data processing	More	Less
Resource consumption	More	Less
Database requirements	None	High write throughput
Reorganization and migration difficulty	High	Lower
Probability of errors	Low	Higher
Balance between speed and reliability	Good	Good
Fixed memory	No	Yes

Table 4.1: Overview of the differences between Lambda and Kappa architectures.

Based on these difference it is not clear what architecture is the overall best solution. Thus, for each new situation, these architectures must be evaluated to find the best fit for the current situation. However, the key differences between them is that the Kappa architecture is about simplicity of development where only one processing framework is needed for all types of data. On the other hand, the Lambda architecture is more fault tolerant and can essentially function in any type of situation, compared to the Kappa architecture that is most effective where active performance of the batch layer is not necessary for meeting the requirements of the system [30].

4.2 Stream Processors

Stream processors are the corner-stone of data streaming, without them the benefit of data streaming would be minimal. Thus, choosing the most optimal processor for a given environment is essential for drawing any value of data streaming. In this section, the stream processors 'Kafka', 'Storm', 'Flink' and 'Spark' will be evaluated based on the requirements established in Chapter 1. It is assumed that general knowledge about these stream processors are known by now, and it's not necessary to be repeated in this section. Furthermore, some of these requirements have been

narrowed down into metrical measurements which will aid in the evaluation process. Although, some general measurements have been used. While the others are specifically tailored metrics to fit in Corporater's innovation strategy:

- Latency
- Throughput
- Resource Consumption
- API
- Ease of Programming
- Input Support
- Output Support
- Ease of Setup
- Architecture Support

Note that, not all on this list are based upon the requirements. This is done to give a broader understanding of the stream processors differences. Further in this section there will be introduced combinations of these metrics that emphasizes and highlights their differences even more

the stream processors differences. In these combinations of metrics, each metric will be introduced separately. After this introduction, each of the stream processors will be evaluated against the metric at hand. When all of the metrics are introduced and evaluated, there will be a summary that highlights the best performer in each combination of metrics.

4.2.1 API and Architecture Support

For many businesses it is important to combine stream and batch data. In fact, for some of them it is essential to have batch enrichments on the data stream. In these situations, it can be valuable to have good API support for batch data, preferably within the same API as the stream. The API metric will highlight this factor and see what type of stream processors have the closest connection to batch data through their API. Moreover, it is possible that some of them do not even have a supporting API for batch data, which in some cases can be quite crucial to have for a business.

Some of our processors do not support batch processing, these are Kafka and Storm which do not have support for batch data in their respective API's [31, 32]. Their focus are solely on data streams in which the API is developed well for. Although, it is still possible to use batching data in these systems, the batch would then have to be converted into a data stream. However, it would be lacking efficiency compared to a processor that supports batch data, which is more optimized for high throughput processing. Furthermore, in terms of the API metric, these stream processors are in the lower region of the spectrum.

Above Kafka and Storm is Flink, which have batch data support but do not have it in the same API as streaming data. Thus, programmers using Flink with both batch and stream processing would need two separate API's to be able to use both batch

and streaming data [33]. Flink is much closer to a more unified API than Kafka and Storm, however batch and streaming are still divided which is why Flink is considered as mid-tier in the API metric.

At the top of these processors is Spark, which has unified both batch and streaming data under one API [34]. Although, there are still some lingering differences between the usages of batch and streaming, it is still currently the closest to a unified API of them all. An example of these differences can be the command for reading batch data 'read' versus the command for reading stream data 'readStream'. Another example is that not all calculations on batch data can be done on streaming data, such as 'pivot' which is supported for batch data but not streaming data. However, pivoting is not supported by any other stream processors to this day. Since Spark is the closest to a unified API of all these stream processors, it is considered to be in the top-tire of the API metric.

In terms of Architecture, the processors needs to both support batch and streaming data in order to implement the Lambda architecture fully. On the other hand, batch support is not required for the Kappa architecture. Thus, the Kappa architecture is less strict for stream processors which easily implementable for them. Furthermore, if a stream processor implements the Lambda architecture it also implements the Kappa architecture since it only need to use the streaming part of their processor to be compatible. This necessarily does not mean that the API metric rating is high for all stream processors that implements the Lambda architecture, however if the API rating is high then most likely this processor supports the Lambda architecture. This is shown in Figure 4.1.

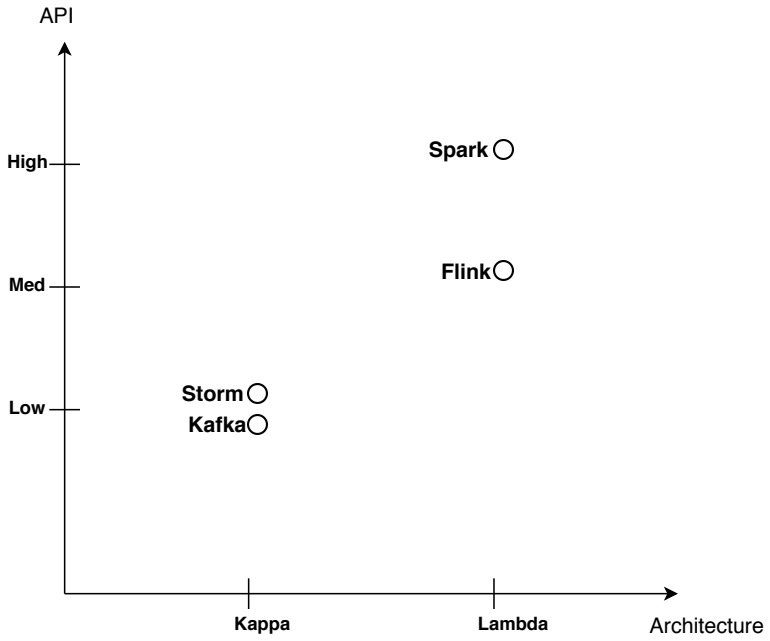


Figure 4.1: API versus Architecture Support.

From this figure, it is possible to see that both Kafka and Storm is in the lower tiers in terms of both Architectural support and API. This means that they have both limitation on Architecture and API which other processors don't, and is rated poorly in this combination of metrics. On the other side of the spectrum is Flink and Spark. Both of them support both batch and stream processing in one environment, thus implementing the Lambda architecture. However, Flink has lower API rating which gives Spark a higher overall rating. Thus, Spark is clearly the better performer in this combination of metrics.

4.2.2 Ease of Setup and Programming

To setup and optimize a stream processor might take considerable amount of time. For some businesses this is not an option. They require a quick deployment without much time consuming optimizations. In these situations choosing a stream processor with a simple setup is crucial. Additionally, this could also reduce costs for the business, by curtailing the developers workload enabling a more rapid implementation. The Ease of Setup metric will highlight these factors in each of our stream processors. Furthermore, by highlighting these factors it should be clear what the advantages and disadvantages are in each of the processors.

One of the most time consuming and complex setups is Kafka. This processor requires multiple 'Command Line Interfaces' (CLIs) in order to implement a simple setup. This is because of the low-level nature of Kafka, which allows for control and tuning capabilities that other stream processors cannot. Furthermore, Kafka does also require a third party software called Zookeeper that coordinates the jobs for the brokers. Utilizing a third party software in this way increases the complexity for the user. Thus, the user must be able to handle both Kafka and Zookeeper in order to run the stream processor. Additionally, since Kafka is low-level, it can be integrated in essentially any deployment technology, such as Docker, Mesos or YARN [35]. Although, in terms of this metric, it is considered to be in the lower-tier.

A processor which is similar to Kafka in this metric is Storm. It is low-level, where multiple CLIs are required for a simple setup. Another similarity to Kafka is the use of Zookeeper. Here it is also used as its coordinator in the cluster, which increases the complexity for the user in the same way as for Kafka. However, with the use of topologies in Storm, allows the processor to do some of the optimization of the running calculations. Still not much is lost in terms of control and tuning from the users perspective. Additionally, the large user base and several of example setups, allow for an easier way of implementation than Kafka. Because of these reasons, Storm can be considered to be in the mid-tier of this metric.

Spark on the other hand, is less complex than Storm in terms of setup, which gives it an edge in this metric. Spark is a cluster framework, where not much knowledge about the underlying system is required for a simple setup. Furthermore, Spark does not require any third party system in its cluster, which is something

the previous processors utilized. Additionally, it only requires one CLI, in which it is possible to also develop all data stream handling. This is something none other of our stream processors have. However, it is still possible to develop in the traditional way. On the other hand, the complexity of configuration could have been improved, since it requires the user to know which nodes are the slaves and which node is the master. Thus, based on these reasons, Spark is considered to be just in the middle of the top and mid-tier of this metric.

Flink is quite like Spark in this metric. This is because, they both are cluster frameworks, where not much knowledge about the system is required by the user. However, Flink is a little less complex than Spark, since it requires less from the user in terms of configuration. In the same manner as Spark, Flink only requires one CLI and do not use any third party systems. Furthermore, this processor requires least amount of configuration and setup implementation of our processors. Thus, it is considered to be in the top-tier of this metric. As a summary, these processors are shown in the Figure 4.2, which shows the processors in their respective tiers.

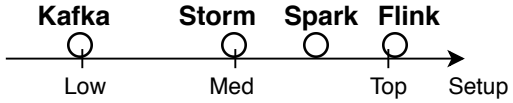


Figure 4.2: Ease of Setup summary.

Businesses with multiple programmers developing and improving a system requires the programming to be as easy as possible. This is not only needed to have quick deployment of the software, but also to have a good and understandable code that a developer can quickly get into. Furthermore, understandable code strengthens the durability of the system, where fewer confusions and errors can be made. By choosing a processor with less complexity in the code can reduce these issues, which is what the Ease of Programming metric aims to measure. During the setup phase of each environment there were also some programming. Not only was the programming used to set up the environment but also to evaluate this metric in each respective processor.

One of the lowest performers in this metric is Kafka. As mentioned before, this processor is quite low-level, which requires the user to explicitly define how the flow of data happens. Thus, much more programming is required in order to describe the whole system rather then only defining the calculations. A simple code example of such an implementation is illustrated in Listing 4.1.

```
public class Producer(){
  private static int port = 9999;
  private static String hostname = "localhost";

  public static void main(String[] args) {
    Socket socket = new Socket(hostname, port);
    BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
    Properties props = new Properties();
```

```
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9000");
        props.put(ProducerConfig.CLIENT_ID_CONFIG, "kafka");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            LongSerializer.class.getName());
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName());
    Producer<Long, String> producer = new KafkaProducer<>(props);

    String line;
    Long index = 0;
    while((line = in.readLine()) != null){
      producer.send(new ProducerRecord<>("simulation", index++, line.trim()));
    }
    producer.close();
  }
}

public class Consumer(){
  public static void main(String[] args) {
    Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "localhost:9000");
        props.put(ProducerConfig.CLIENT_ID_CONFIG, "kafka");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            LongSerializer.class.getName());
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName());
    Consumer<Long, String> consumer = new KafkaConsumer<>(props);

    consumer.subscribe(Collections.singletonList("simulation"));
    while(true) {
      ConsumerRecords<Long,String> consumerRecords = consumer.poll(1000);
      if(consumerRecords.count()==0){
        break;
      }
      consumerRecords.forEach(record -> {
        System.out.printLine(record.key() + " " + record.value());
      });
    }
    consumer.close();
  }
}
```

Listing 4.1: Kafka example.

This listing shows two Java classes, one for writing the stream to a Kafka topic called 'simulation' and one for reading from the same topic. In this example, no computations are done on the records, which means that the data is only flowing through Kafka. In order to incorporate efficient computations, a third class is required, which uses the Kafka Streams library. However, it is not needed in order to illustrate how complex a simple topology is in Kafka. Moreover, there are few other libraries that can be utilized in Kafka. For example, Machine Learning and Graph Analytics are not supported. On the other hand, there is support for SQL

and a connection API which enables connection to different sources. Additionally, because of its low-level approach, Kafka is compatible with several different programming languages. This can allow the developers to program in their preferred language, which can increase the productivity of the development process. For these reasons Kafka is considered to be in the lower-tier of this metric.

A quite similar processor in this metric as well is Storm. Thus, it is also one of the low-level processors where more programming is required. However, Storm requires more structure in its programming. This is done through the Topology analogy, where there exists a main file with all 'Spouts' and 'Bolts' gathered to form a topology. This is code structure is illustrated in Listing 4.2.

```
public class Spout extends BaseRichSpout {
    private SpoutOutputCollector collector;
    private BufferedReader in;
    public void open(Map map, TopologyContext ctx, SpoutOutputCollector collector) {
        this.collector = collector;
        String host = "localhost";
        int port = 9999;
        try {
            Socket s = new Socket(host, port);
            in = new BufferedReader(
                    new InputStreamReader(s.getInputStream()));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public void nextTuple() {
        try {
            String jsonString = in.readLine();

            if (jsonString == null) {
                throw new SocketException();
            }

            collector.emit(new Values(jsonString));

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public void declareOutputFields(OutputFieldsDeclarer decl) {
        decl.declare(new Fields("field"));
    }
}
public class Bolt extends BaseBasicBolt {
    private int counter = 1;
    public void execute(Tuple tuple, BasicOutputCollector basicOutputCollector) {
        String text = tuple.getString(0);
        basicOutputCollector.emit(new Values(text));
        System.out.println(String.valueOf(counter) + ": " + text);
        counter++;
    }
```

```
    public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {
        outputFieldsDeclarer.declare(new Fields("field"));
    }
}
public class Topology {
    public static void main(String[] args) {
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("Spout", new Spout());
        builder.setBolt("Bolt", new Bolt()).shuffleGrouping("Spout");

        Config config = new Config();
        config.setNumWorkers(20);
        config.setMaxSpoutPending(5000);

        StormSubmitter stormSubmitter = new StormSubmitter();
        try {
            stormSubmitter.submitTopology("simulation", config, builder.createTopology());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Listing 4.2: Storm example.

In this listing there are three Java classes: Spout, Bolt and Topology. The Spout is writing in to the topology and the Bolt is reading from the topology. Furthermore, the Topology class creates the connection from Spout to Bolt. Compared to Kafka there are more code and classes required to do a similar type of job. However, if more calculations are required, it wouldn't necessarily need another class such as Kafka. On the other hand, with only the support of streaming SQL, Storm contains the least amount of supporting libraries, which can make it less useful in some situations. Another similarity to Kafka is its multitude of supporting programming languages, where virtually any programming language can be utilized. Storm is on the lower-tier of this metric, based on the aforementioned advantages and disadvantages.

Flink on the other hand, handles the topology and flow for the user. This makes Flink one of the higher-level processors for streams, where much less programming is required by the developer. Since all of the optimisation and topology logic is handled by Flink, only the calculations are needed to run an implementation. One such implementation is illustrated in Listing 4.3.

```
    public class Flink {
        public static void main(String[] args) throws Exception {
            final StreamExecutionEnvironment env = StreamExecutionEnvironment.
                getExecutionEnvironment();
            DataStream<String> stream = env.socketTextStream("localhost", 9999);
            DataStream<String> stream = applyCalculations(stream);
            stream.print();
            env.execute();
        }
```

```
}
```

Listing 4.3: Flink example.

Based on this listing, it is clear how much simpler programming is in Flink compared to Kafka and Storm. With only one variable it allows users to create and access the cluster. Thus, only one Java class is actually required to implement Flink. This is much smaller than Kafka and Storm which required at least two or three classes for an implementation. However, there are minimal language support for programmers. In fact, Flink supports only Scala, Java and Python. Even though these are the few of the most essential languages in programming, it is limited in comparison to the previous processors. Additionally, similar low library support is provided by this processor. Where only the CEP and SQL library is supported for streaming. Even though the internal programming is at a high level, the other functionalities around it is quite limited. Which is why Flink is considered to be in the mid-level of this metric.

Spark is in the same level of programming as Flink. Thus, programming happens in a high-level and most of the optimisation and topology logic is handled by Spark. Given that Spark is in the same level as Flink, it is logical that similar amount of code required by Flink is also required by Spark. An example of such an implementation of Spark is shown Listing 4.4.

```
public class Main {
    public static void main(String[] args) throws IOException {
        SparkSession spark = SparkSession.builder()
            .master("local[3]")
            .appName("simulation")
            .getOrCreate();
        Dataset<Row> stream = spark.readStream()
            .format("socket")
            .option("host", "localhost")
            .option("port", 9999)
            .load();
        Dataset<Row> stream = applyCalculations(stream);
        stream.writeStream()
            .outputMode("append")
            .format("console")
            .start()
            .awaitTermination();
    }
}
```

Listing 4.4: Spark example.

At first glance this listing might not look similar to Flink. However, with a closer look it is possible to detect similar structure between the two. For instance, both of them are initiating an environment/session and reading the stream through the environment/session. Furthermore, both of them are applying calculations in the

same manner, and writes the output through the stream variable. Although, in Spark's case, the code is more generalized where the method input parameters dictates the behaviour of the method at a greater scale than in Flink. Furthermore, with this generalization, it is clear that more inputs are required by the user in order to do the same job as Flink, which might increase the programming time required to implement it. However, there are less methods required to keep track on which can reduce the programming time. One could argue for either Flink or Spark on this topic, where both of them are good contenders for a high rating in the Ease of Programming metric. However, Spark has an edge on Flink in terms of programming language and library support. This processor supports Java, Python, Scala, SQL and R as programming languages, which is slightly more supportive than Flink. While on the library support, Spark is the only of our stream processors that supports Machine Learning and Graphical Analytics out of the box. Which is why Spark is considered to be in the top-tier for this metric.

To summarize this metric, we present our findings in a chart. This chart is a combination with the Ease of Setup metric, which allows for a better overview and similarities between them. The chart is shown in Figure 4.3.

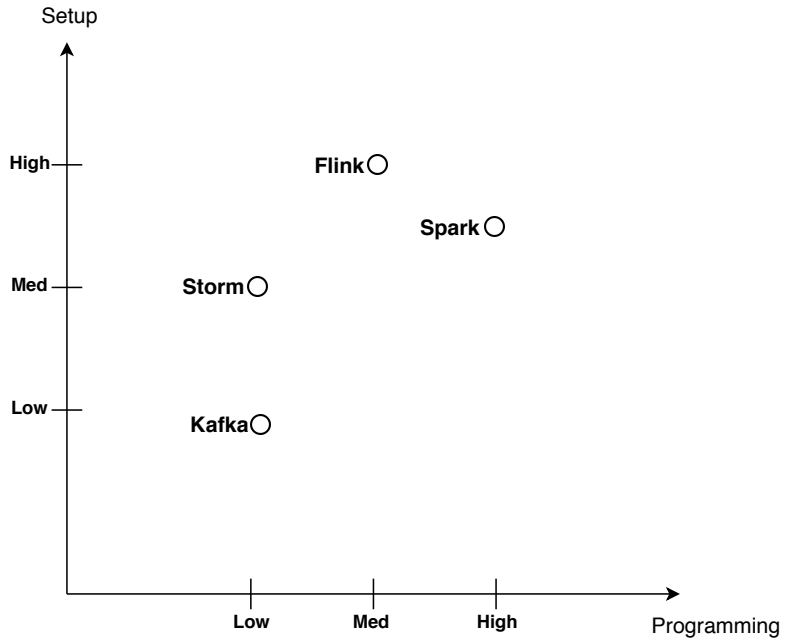


Figure 4.3: Ease of Setup versus Ease of Programming.

From this figure, it is possible to see that Spark and Flink have high measurements in both of these metrics. Both of them are rated quite high in both of the metrics. However, Spark has an edge in benefits that often makes it worth the little extra time and complexity with setup. Thus, Spark is the better performer in this combination of metrics. On the other hand, for situations where more control of the pipeline is required, then neither Flink or Spark is recommended. These processors

handle more of the pipeline for the programmer. Thus, it would be working against these systems in such situations. On the other hand, a more beneficial choice can be either Kafka or Storm. Since both of them use a low-level programming approach that enables developers to control the pipeline.

4.2.3 Latency, Throughput and Resource Consumption

A commonality between the Latency, Throughput and Resource Consumption metrics, is that they are all numerical measurements that can be found in Table 2.1. Furthermore, it is possible that some of these measurements are not up to date with the most resent release. Although, we are still able to categorize them and find the best performers given that they are somewhere around these previous benchmarks.

Latency is one of the most important metrics within stream processing. This is because it shows how real-time a stream processor actually is. For instance, there are many businesses out there that rely heavily on getting the newest information as quickly as possible. For these businesses having lower latency than their contenders can give them an edge in a brutal market. Latency is measured in the time difference between when an event enters the stream processor until it exits. Not all stream processors perform the same way in this metric. This might be because of how their internal architecture is structured or how each event is processed. Thus, some processors are able to achieve millisecond latencies where others have seconds.

Spark is one of the latter once, which have latencies around the 1 second mark. Mostly this is due to the basic principle in Spark, which considers streaming as a special case of batching. This means that all streaming data is converted into batches, in fact, micro-batches that contains multiple events. Thus, when one event exits, multiple other events also exit. Spark is considered as one of the stream processors with the highest latencies, and is in the lower-tier of this metric.

Above Spark is Kafka and Flink, these processors are able to achieve latencies around the 100ms mark. In these processors, streaming is not considered as a special case of batching, and each event is processed as they arrive. Additionally, in order to provide fault tolerance and messaging guarantees such as 'Exactly-once Semantic', these processors use backwards acknowledgements and check-pointing, which increases overhead and processing time, thus increasing latency. The achievable latencies of Kafka and Flink puts them in the mid-tier of this metric, which can be good enough for most use cases.

At the top is Storm, which can provide latencies under the 100ms mark. Similarly to Flink and Kafka, Storm does not consider streaming as a special case of batch allowing it to achieve subsecond latencies. Additionally, Storm can turn off backwards acknowledgements that allows it to outperform Flink and Kafka, which is why it is considered as top-tier processor in this metric. However, this low latency comes at a cost, which is little to non fault tolerance and no messaging guaran-

tees. These disadvantages may be too high of a cost for businesses to pay, which is why it is important to be aware of these. Figure 4.4 shows each processor in their respective tiers, and serves as a summary of this metric.

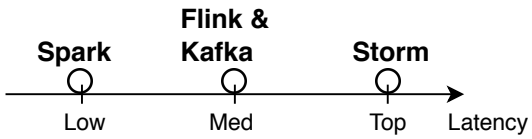


Figure 4.4: Latency metric summary.

A different, yet just as important, metric is the Throughput metric. This metric illustrates how many events can be pushed through the processor at once. Also this metric can be quite valuable from a business perspective, since it essentially shows how much value the business can draw out of the processor at once. Thus, some businesses might value higher throughput rather than lower latency. Similarly to the Latency metric, not all processors perform equally in this metric. These different performances will be highlighted in a similar manner as the previous metric.

One of the lowest performers in this category is Kafka. It has a throughput around 100-800K records per second, which is low compared to our other processors. Not only is this because of its inner architecture but also because of a different metric called Resource Consumption, which will be introduced later in this section. Kafka was one of the better performers in the previous metric, however in this metric it is considered in the lower-tier.

A processor with better performance than Kafka in this metric is Storm, which is able to achieve over 1 million records per second. Storm also had the highest rating in the Latency metric, which is quite impressive to additionally have this high throughput. Flink was also quite close to Storm in the Latency metric, however in this metric Flink outperforms Storm. Flink can achieve a throughput around 10-40 million records per second, which in some cases can make up for not having the same or higher latency as Storm. Furthermore, both Storm and Flink can be considered to be around the mid-tier level for this metric.

The processor with the highest throughput capabilities is Spark, which is able to achieve a throughput around 50-60 million records per second. Considering that it was the worst performer in the previous metric, it is clear that throughput is a larger focus for Spark. Furthermore, from an architectural standpoint it is logical that Spark is at the top of this metric, since Spark is based on batch processing which is more optimized to handle large throughputs. Thus, Spark is considered to be at the top-tier of this metric. A summary of these performances are shown in Figure 4.5.

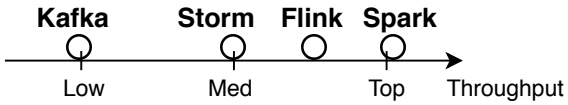


Figure 4.5: Throughput metric summary.

The last metric in this combination is the Resource Consumption metric. This metric is not one of the most focused metrics in other benchmarks. However it should be just as important as the other two metrics. For instance, a business can reduce the operational budget of a stream processor by choosing a processor that uses less resources to do the same job as another stream processor with much higher resource consumption. Furthermore, these savings can be used in other systems or projects that can give more value to the business. In the same manner as the previous metrics, each processor must be evaluated to find the best performer.

The highest consumer of resources is Flink, which was one of the highest performers in the previous metrics combined. This seems to be rather fitting that also Flink is a low performer at one of these metrics. Furthermore, it also hints to the existence of a trade-off between the three, where one metric is hampered for the benefit of the others. Based on this high consumption of resources, Flink is considered to be in the lower-tier.

Two processors that are similar in this metric is Storm and Spark, which both outperforms Flink. However, the leading performance of our last processor makes them both to be in the mid-tier of this metric. Interestingly, they perform well in separate other metrics. Spark performs well in the Throughput metric, whereas Storm performs well in the Latency metric. This further hints to the existence of a trade-off between the three metrics.

Kafka on the other hand performs excellent in this metric. Although, the performance is mediocre in the other metrics, it makes up for it here. Thus, it is clear that Kafka has chosen to focus on keeping the resource consumption to a minimum, which is why the performance in the other metrics are compromised. However, for this metric, it is considered as part of the top-tier. Additionally, this does also strengthen the claim that there exists a trade-off between these three metrics.

Based on these evaluations a radar chart was created to give a better overview of these metrics and show how they affect each other. This chart is illustrated in Figure 4.6.

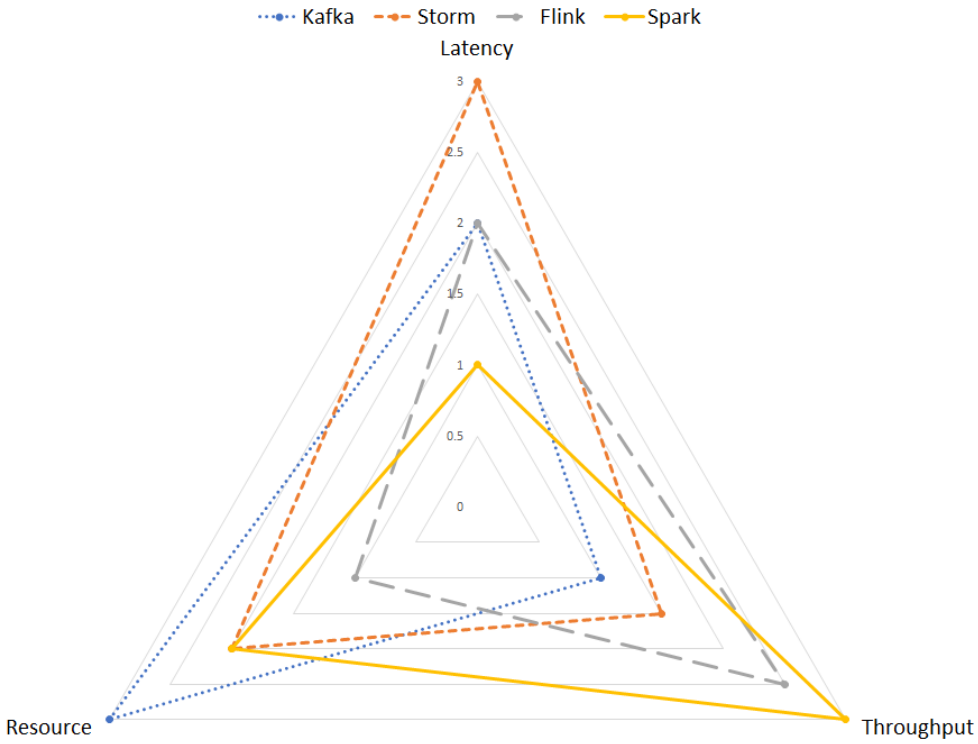


Figure 4.6: Overview over the metrics.

From this figure it is clear that there is no stream processor that outperforms all the other processors in every single metric, and one would have to select which of these are most important for each situation. For instance, if a lightweight stream processor with little resource consumption is needed then Kafka might be the best solution. Another scenario could be if a stream processor needs the lowest possible latency, then Storm might be the best solution. One could also look at it the other way around and find which of these metrics are least important for the given scenario. For instance, if resource consumption is least important, then probably Flink is the best option. Alternatively, if latency is the least important then possibly Spark is the best solution.

4.2.4 Input and Output Support

When a business is choosing a stream processor, some may need many connections to other systems. For quick deployment in these situations, it is crucial to have built-in input and output connectors in order to lessen implementation time. However, few stream processors support this, and even less have a broad spectrum of these. In this part, each processor will be evaluated based on how many input and output connectors they have. Later these findings are presented in a graph, which illustrates clearly what each processor contributes with.

One of our most flexible stream processors is Kafka, which is able to connect to everything from databases and HDFS to stream sources and applications [10]. However, connecting to these systems requires a developer to create a 'Producer' for input data and a 'Consumer' for output data. Thus there are no built in connectors in Kafka, except for a console producer and consumer, which allows reads and writes to happen through the console [10]. A similar system in this regard is Storm, which requires an implementation of a 'Spout' for the input data, and a 'Bolt' for the output data. However, since it is one of our most used stream processors, many of these connectors are already implemented in other open source systems which allows for reuse in other implementations. Additionally, Storm developers have different guides and examples on how to create connections to all kinds of systems such as HDFS, Amazon Kinesis and Elasticsearch [36]. Still, this would have to be implemented in each new Storm instance, thus there are no built-in connectors in this processor as well. In conclusion, both Kafka and Storm is considered to be around the lower-tier of both the input and output metric.

A processor that outperforms both Kafka and Storm in this metric is Spark, which has input and output support for Kafka and File [34]. Furthermore, there is also input support for Socket connections. Spark does not have the largest support of built-in connections. However, there is at least some that can be utilized out of the box, which is more than what the previous stream processors can provide. Thus, Spark is considered to be around the mid-tier for both the input and output metric.

A dominant processor in these metrics is Flink, which has the largest support in both input and output support. This is achieved through a package called 'Connectors' which is included in the initial installation of Flink. Furthermore, this package includes connectors such as Elasticsearch, RabbitMQ and Amazon Kinesis, which allows for easy setup and connection that can reduce time and resource. Thus, with this package, Flink is considered to be at the top-tier of both of the metrics in this evaluation. Moreover, the total evaluation is illustrated more clearly in Figure 4.7.

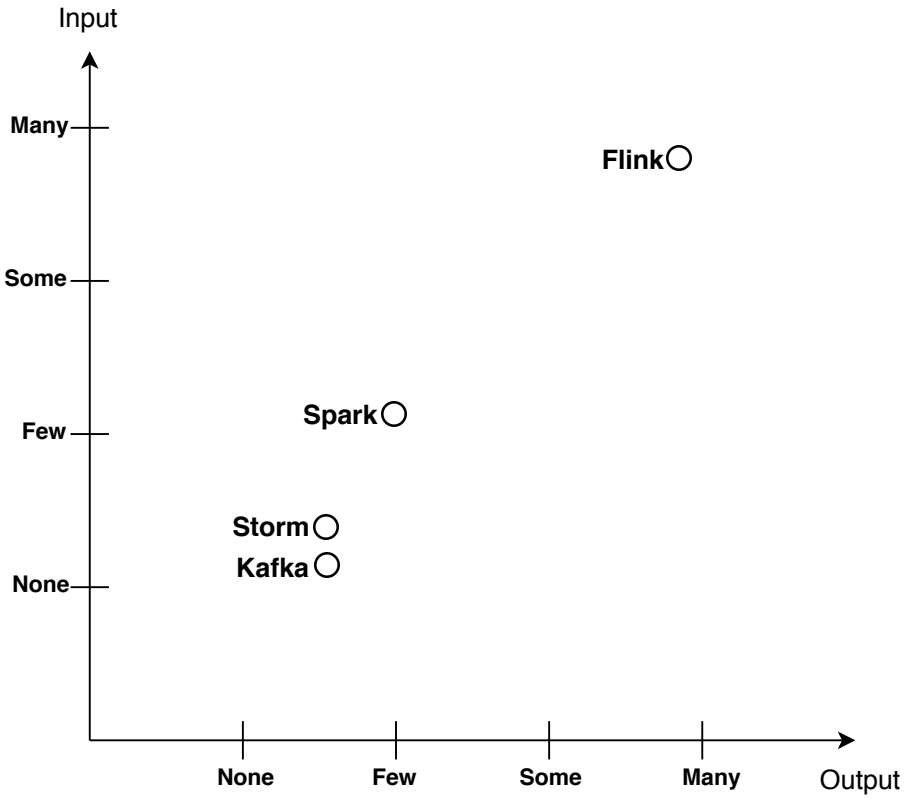


Figure 4.7: Input and Output support.

In this figure, both the Input and Output metrics have their own reference line, which refers to how many inputs and outputs a processor has built-in to their systems. Furthermore, it shows that Kafka and Storm are the lowest performers and are tightly followed by Spark which has a few connectors in both categories. On the other side is Flink, which has the largest support of connectors to other systems, thus making it a clear winner in this combination of metrics. However, even though both Kafka and Storm are among the lowest performers, it can still be possible to have a quick implementation of them. This is because of the many open source implementations and integrations for them, which generally contains the current most used inputs and outputs. Additionally, since Spark has a built-in connection to Kafka, it can also benefit from Kafka's open source solutions.

4.3 Summary and Recommendations

In this chapter, there have been two different evaluations, one evaluation of architectures and one of stream processors. In the architecture evaluation, the architectures Kappa and Lambda were compared with each other. These architectures were quite similar, however, the main difference is that the Lambda architecture has a separate Batch layer whereas the Kappa architecture only has a Speed layer. Fur-

thermore, Kappa focuses more around simplicity of development, whereas Lambda is more fault tolerant. In the stream processors evaluation, nine different metrics were introduced. Most of them are of importance to Corporater, whereas the others are provided for a better overall view of the stream processors. These metrics functioned as a separator of all the stream processors differences. Furthermore, this approach narrowed down the spectrum of differences into more manageable sizes, which allowed to focus on each important difference one at a time.

These evaluations provides us with enough knowledge to recommend a solution for Corporater. Not only are the recommendation based on their requirements introduced in Chapter 1, but also their scenario as well. Furthermore, not all requirements by Corporater are intended for this chapter. Thus, only the once relevant will be discussed. Moreover, there are two requirements that are covered on all possible system implementations. The first one is that the system should not need any other licensing from other vendors. Since all systems mentioned here are open source, this requirement is implicitly fulfilled. The second one is that the system should support both Linux and Windows operating systems. Additionally, the stream processor should also support cloud deployment. From the processors respective websites [10, 33, 32, 14], it is shown that this requirement is also fulfilled for all possible system implementations.

In the same manner as the first requirement, this is also implicitly fulfilled. Further in this section, each evaluation will be compared to Corporater's remaining requirements and scenario.

4.3.1 Architecture Recommendation

One evaluation without a requirement is the Architecture evaluation. Based on the survey, it was not possible to draw out such a requirement. However, from their scenario it is possible to deduce a recommendation. By now it is known that Corporate delivers a software called BMP. This software is integrated with batch processing, which can be considered as a Batch layer. Furthermore, it is intended that data stream processing is to be included in this software. Thus, in order for a smooth integration with the rest of the system, it is recommended to use the Lambda architecture.

If the Kappa architecture was implemented completely in this scenario. It would require all of the batch integrations to be ported over to the Speed layer. This would mean a revolutionary change in their system, which will come at a huge cost to the company. Furthermore, using Kappa would also mean putting a requirement on customers databases, that would need the ability to handle high write throughputs. Although, by choosing the Lambda architecture it does not mean that the Kappa architecture can never be implemented in this system. This architecture can always be implemented in the future. By first using the Lambda architecture the conversion can happen more gradually, where piece by piece of the Batch layer is converted into the Speed layer. Thus, the Lambda architecture is probably the best

choice for the present and future of Corporater.

4.3.2 Stream Processor Recommendation

There are multiple requirements from Corporater for the stream processor. All these requirements have been covered by the evaluation of the different metrics. Thus for this recommendation each requirement will be introduced separately. In order to find the best fit for the requirement, the corresponding metrics will be utilized. Further, all these requirements will be combined into a final recommendation for Corporater.

One of these requirements is the large input and output support of the stream processor. This requirement is based upon the already large variety of customers that Corporater has. Thus, a useful stream processor for them should be able to support all these customers. From the evaluation of input and output support, there were quite few highly supportive processors. In fact, Flink might be the only processor that can support all the customers of Corporater. On the other hand, it is possible to use the other stream processors by discovering their respective library extensions. This enables them to have the same or higher support than Flink. However, these processors might need more development time in order to be properly implemented.

Two other requirements for the stream processor are as many calculations as possible, and ability of advance aggregations. These requirements are based on the flexibility of the existing BMP, that can be utilized in all different kinds of use cases. Thus, the stream processor should do the same. There is one metric that can be used for these requirements, this is the Ease of Programming metric. Not only does this metric show how it is to program in each of the systems but also what calculations and libraries that can be utilized. Spark is the highest rated stream processor in this metric, which makes it an excellent choice in this regard. However, it is more complex to setup compared to Flink, which is important to be aware of. Since it could cost more development time in the setup phase of the implementation.

A high throughput is important for Corporater. This is based on the limited use cases one would have for the BMP with a low throughput stream processor. Additionally, to have a broad spectrum of utilization capabilities would be highly valuable. Thus, this requirement is rated heavily compared to other requirements. The corresponding metric to this requirement is the Throughput metric. This metric shows that both Flink and Spark are highly compatible. However, based on Figure 4.6, they excel in separate other metrics which can be the deciding factor between the two. For Corporater's scenario, it is clear that latency is not a priority in their system. Thus, it is more valuable to chose a processor that is superior in the Resource Consumption metric. Spark has better performance in that metric, thus surpassing Flink in terms of this requirement fulfilment.

The final requirement from Corporater is that the processor needs to have a close

connection to batch data through its API. This requirement is based upon the value of enriching the stream with batched data. Furthermore, it also enables more functionalities to happen between them, such as comparing today's current status with the same day last year. The one metric fitting this requirement is the API metric. From the evaluation of this metric, it is known that Spark is the highest rated processor. This is because of the 'Structured Streaming' library that enables to use the same library on both batch and streaming data. Furthermore, no other stream processor have this support. Thus, for this requirement Spark is the best solution.

There are several factors to consider when choosing a stream processor. Most likely not all of these factors are included in this evaluation. However the most important once for Corporater are present. Based on these factors, the most fitting stream processor for Corporater is Spark. This processor scores the highest in the most valuable metrics. Furthermore, it comes with a lot of extra features that other processors can not compete with. Even though the input and output support is not optimal for this processor, it compensates with being an excellent choice in the other requirements. However, by utilizing another system like Kafka, which Spark has native support to, enables adequate input and output support for the overall system. Additionally, Spark is able to support the Lambda architecture completely. This also enables future batch implementations to be done in Spark. Thus, unifying batch and streaming data, without the need to convert to the Kappa architecture. On the other hand, this does not mean that Spark is the most optimal choice for every scenario. For instance, Spark would not be recommended when latency is important, or when control of the pipeline is required. In these situations it might be best to use either Kafka or Storm. Another example where Spark might not be the best choice, is when ease of setup is important or multiple connections to other systems is highly required out of the box. For this instance Flink might be the best choice. However, for Corporater's scenario Spark is the most beneficial stream processor for them.

Chapter 5

Proof of Concept

From the previous chapter, an architecture and a stream processor have been recommended. In chapter 3: Solution Approach, these two were injected into a prototype. However, this prototype was not tested or analyzed in any way. Thus, it is not known whether Corporater's requirements are fulfilled or not. In this chapter, our prototype will be put to the test through an experiment of a typical scenario for Corporater. This experiment will serve as a proof of concept, and show what is possible to implement in Corporater's system. Before the prototype is tested, there will be a thorough explanation of how this experiment is configured and set up. Additionally, the stream generator will be properly explained, which was only introduced as a generator in previous chapters.

5.1 Stream Generator

Since there was no data available for simulating a data stream, it had to be created. This was done through a generator which is meant to emulate a call center where customers call in for information or help. The call center is handling calls for multiple businesses where all calls go through the same stream. The customer flow is illustrated in Figure 5.1.

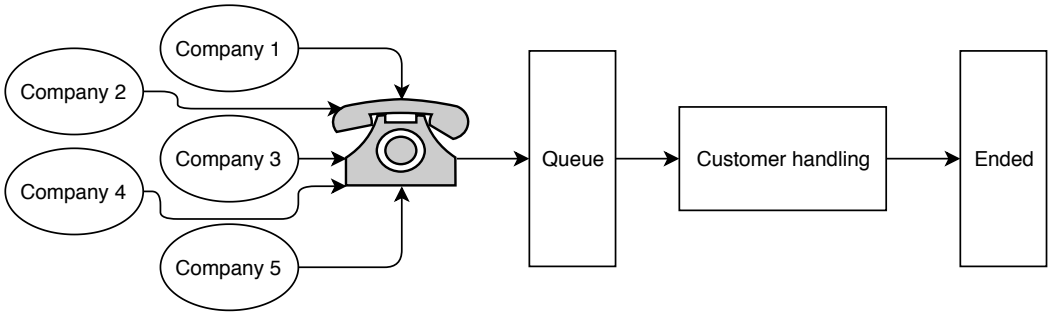


Figure 5.1: Customer Flow of Call Center.

Following Figure 5.1 from the left, shows that a customer must first enter a queue before talking with an operator. Each time the customer enters the queue a timestamp is logged, this happens also when the customer handling process starts and when the phone call has ended. The generator does not have random customers that end their phone call in the queue, which means that all customers are eventually handled. When a customer ends the call an event is created in JSON format which is sent out on a TCP port. This event consists of:

- **PhoneNbr:** Phone number of customer.
- **CompanyID:** ID of the current company.
- **Operators:** Operators on the job.
- **Enter:** Timestamp of customer entering the queue.
- **Handling:** Timestamp of when the customer starts being handled.
- **Ended:** Timestamp of when the phone call ended.

Initially this generator was more complex and realistic, however the events were not happening fast enough. Thus it was re-implemented with a much simpler design which achieved much higher throughput, in addition the throughput became controllable to keep it within manageable parameters of the prototype.

5.2 Experimental Setup

In Chapter 3 a prototype was introduced. This prototype is integrated with the BMP software and provides an end-to-end connection to the data streams. In this section there will be explained an experimental setup of this prototype. This is done through a series of three parts. In the two first parts there will be explained some example configuration of Spark through BMP. Moreover, the first part will focus on the general configuration and setup of the data stream inside Spark. The second part of the explanation is focused around the calculation pipeline, where each mutational step of the stream is explained. The third part explains the overall architecture of the setup and hardware specifications. Furthermore, the stream generator from the previous section will be utilized as a data source for this experiment.

5.2.1 General Configuration Steps

For this experiment the prototype is intended to function as a micro service within BMP. This is not necessarily the only use case for this prototype. It can also function on its own, where for example a simple GUI is created on top for non-developers to configure. However, in this experiment we will utilize Corporater's systems as an example of how this could be done. This also means that Spark will be booted

up together with the BMP, and its startup configurations are already set by a hypothetical system administrator. Thus, when a stream needs to be created we know that Spark is running and only require the stream configuration in order to start calculations on the stream.

To initiate a stream through BMP, it is necessary to create a resource that contains all required configurations for Spark. The general configuration required at this point are shown in the list below.

- **Host & Port**: Defines the location of the data source through IP address and port number, which Spark can connect to and process events. In our case this is the IP address and port number of the stream generator.
- **Stream Name**: Defines the name of the stream. This name is utilized as a table name not only in Spark internals but also in the H2 Serving layer. In our case we will call our stream 'PhoneEvents' as a fitting name for this stream.
- **Stream Sample**: Defines the structure of the stream. This is a necessity for Spark in order to properly convert the JSON string into row format. Another solution for this, is that the user defines the whole structure by hand. However, this might be too much for the user to configure. Additionally, it takes more time than just pasting a sample of the stream. In our case, we use a sample with the same structure as defined in the stream generator.
- **Window Size**: Defines the time window in seconds for grouping events together. In most cases there will be grouping of data in such a system. Thus, it is necessary to have a time window specified in order to prevent never ending groupings.

These configurations are essential in order for Spark to process the stream properly. A potential configurational interface for these metrics and the respective resource inside Configuration Studio are shown in Figure 5.2.

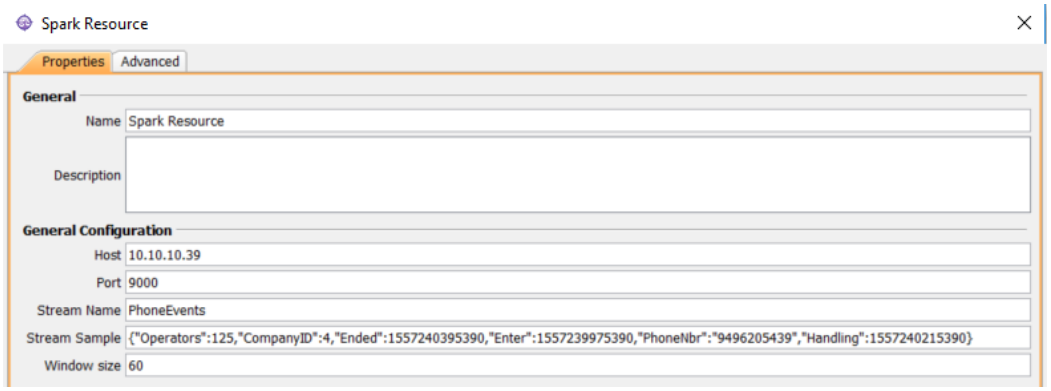


Figure 5.2: General Configuration Example in BMP.

From this figure it is possible to see that it requires no programming in order to setup the most general configuration required for the stream. Using this type of

interface, it would not be necessary to include a developer, rather a simple business user could do this job. In the next part, there will be introduced more configuration examples in the same type of interface, that could tell Spark what to calculate on the stream.

5.2.2 Calculation Pipeline

The calculations in the pipeline are meant to simulate a typical scenario for Corporater. Thus, advanced aggregations and calculations will be one of its main focuses. Based on the call center in the generator, it could be interesting to look at averages of how long a customer stays in the queue and how long the conversations are. Additionally, it is also interesting to see how many operators are present on each average measurement. An illustration of this pipeline is shown in Figure 5.3. Furthermore, mutations that are uncontrollable by regular users will not be shown here, as they are necessary mutations for the system to function properly. An example of such a mutation is the step of converting the JSON string into a table row.

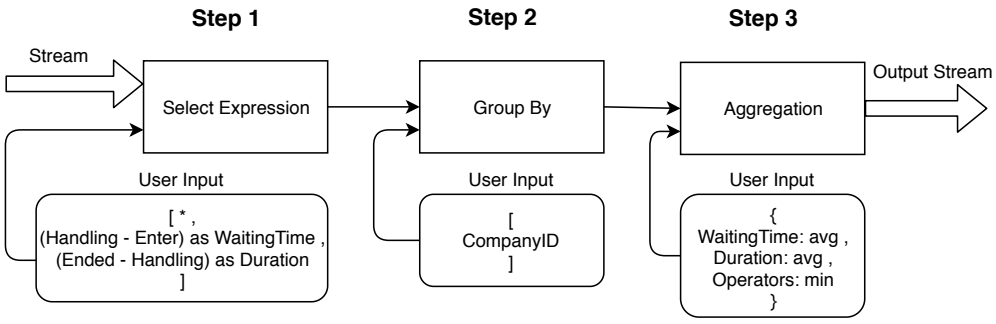


Figure 5.3: Calculation Pipeline.

The first step in this pipeline is the 'Select Expression' method. As mentioned in Chapter 3, this method allows SQL-like queries as input. This is done by converting the different selections from the user into an array that is fed into the method. Furthermore, by selecting the difference between the 'Handling' and 'Enter' parameters, the waiting time for this particular event is created. The same type of difference is also done between 'Ended' and 'Handling' in order to find the phone call duration for the same event. Additionally, to keep all the other columns from the stream with us, we utilize SQL term ' * ' to select all these events. One potential configuration can be created through the interface shown in Figure 5.4.

Figure 5.4: Select Expression Example in BMP.

The second step is a 'Group By' method. This method allows the user to group rows together based on one or more columns. For this case, we want to group by the CompanyID's such that this can further be separated into different tables and charts for each respective company. Additionally, a second grouping is happening under the covers. This is the window grouping that prevents never ending grouping calculations. A potential configuration can be created through the interface shown in Figure 5.5.

Figure 5.5: Group By Example in BMP.

In order to complete the grouping, some aggregations are required. This is covered by the third step in our figure. In this step, the user provides the aggregations of desire. These are further converted into a hash map that is fed into the aggregation. For our case, we want to have the average of the waiting time and phone call duration. Additionally, we include the minimum of operators for this grouping that provides the number of operators on each average measurement. One potential configuration can be created through the interface shown in Figure 5.6.

Figure 5.6: Aggregation Example in BMP.

Based on these calculational steps, we have created the measurements of interest. Additionally, a potential way to configure these calculations has been shown solely using the BMP software. This means that it can be possible for regular business users to configure data streams through a simple GUI, where no programming is required.

5.2.3 Overall Setup and Hardware Specifications

Achieving a certain performance of the system is not a requirement from Corporater. Thus, simplicity of the overall setup is chosen for this particular setup. An illustration of the running nodes in this prototype is shown in Figure 5.7.

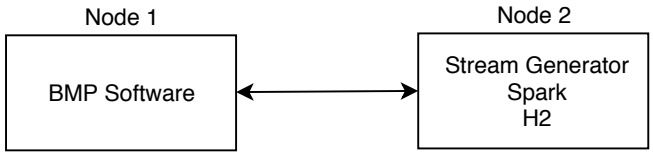


Figure 5.7: Overall Setup.

In this figure, there are two nodes. The first one is running the BMP software and the second one is running the stream generator, Spark as the stream processor and H2 as the Serving layer. This setup mimics a potential setup for Corporater, where the BMP is not concerned about how the data stream is processed and only needs to send configuration to the processor. Furthermore, these two nodes are not the same in terms of hardware. This is illustrated in Table 5.1.

	Node 1	Node 2
RAM	16 GB	32 GB
Cores	2	4
Base Frequency	2.4 GHz	3.6 GHz
Operating System	Windows	Windows

Table 5.1: Hardware specifications of nodes.

This table shows the specifications of the two nodes shown in 5.7. Based on this table, it is logical that Node 2 has more processes running since it is the most powerful computer of the two. With this setup, we are able to achieve a throughput of 300 records per second, which is more than enough for a testing environment.

5.3 Experimental Result

At the end of the calculation pipeline, the calculated events are sent to the H2 database. As mentioned in previous chapters, this database serves as the Serving layer for our prototype, and is the connection between the stream processor and BMP. Thus, when the events are sent to H2 it is also available to be captured by the BMP. Further in this section there will be described how these events are captured and further mutated into visible objects in the BMP web interface.

In order to capture the Serving layer's events a connection must be established between the two. This is done through a resource in Configuration Studio called SQL resource. An implementation of this resource and its respective inputs are shown in Figure 5.8.

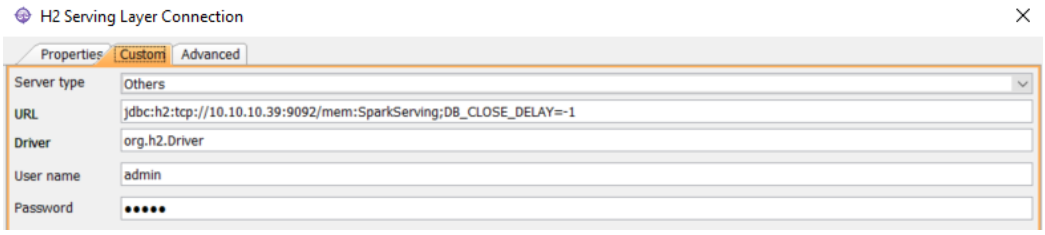


Figure 5.8: Connection to Serving Layer.

From this figure, it is possible to see that only the most basic connection configuration is required in order to connect to H2. This connection can be utilized by other objects to retrieve desired information from the Serving layer through SQL queries. Thus, having a connection to the real-time calculated values from Spark. We created one such object which we named 'Stream Table'. This is a tailored table designed to read the Spark entries in H2. An implementation of this object is shown in Figure 5.9.

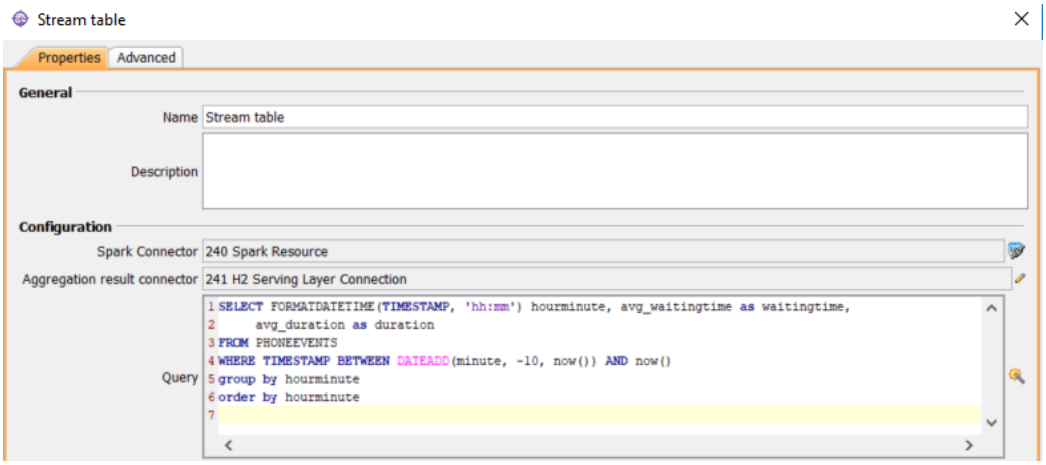


Figure 5.9: Stream Table Object Configuration.

This figure shows a typical Stream Table configuration. In this figure there are three configuration columns. The two first columns are the required resources that needs to be setup in order to read from the Serving layer. The third column is the SQL query that is sent to H2 though the respective resource connection. In our case, we want the last 10 minutes of our two calculated averages 'Waiting Time' and 'Duration' which is shown in the Query input field. With this query, the Serving layer will respond with the requested table. Moreover, this table serves as a reference point for displayable objects in the web. These objects can show this information in several different charts or tables that can aid the business experts to find the use full information quickly. Two such objects are created in Configuration Studio to read from this table. These objects are shown in the web interface illustrated in Figures 5.10 5.11.

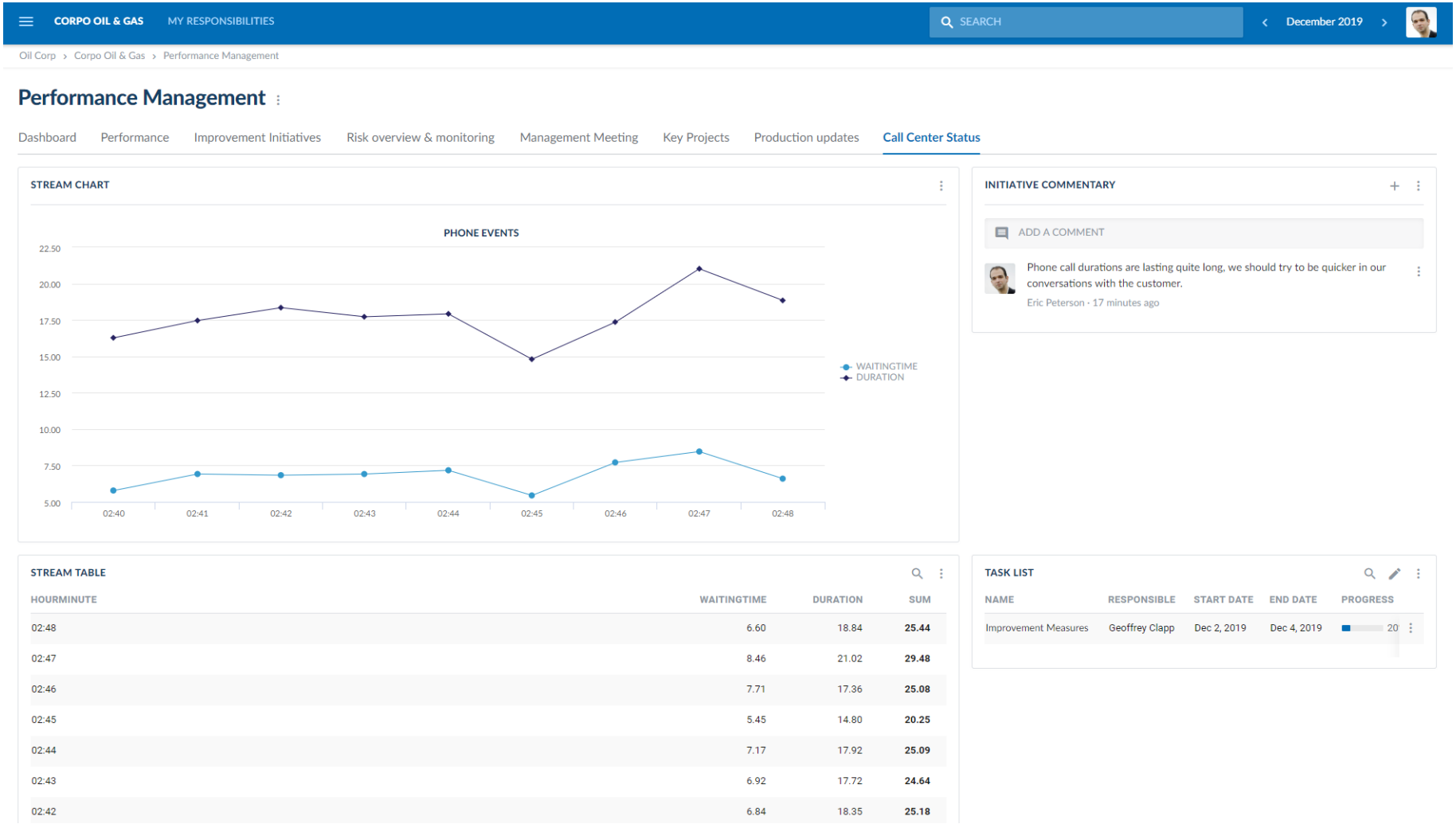


Figure 5.10: First Result.

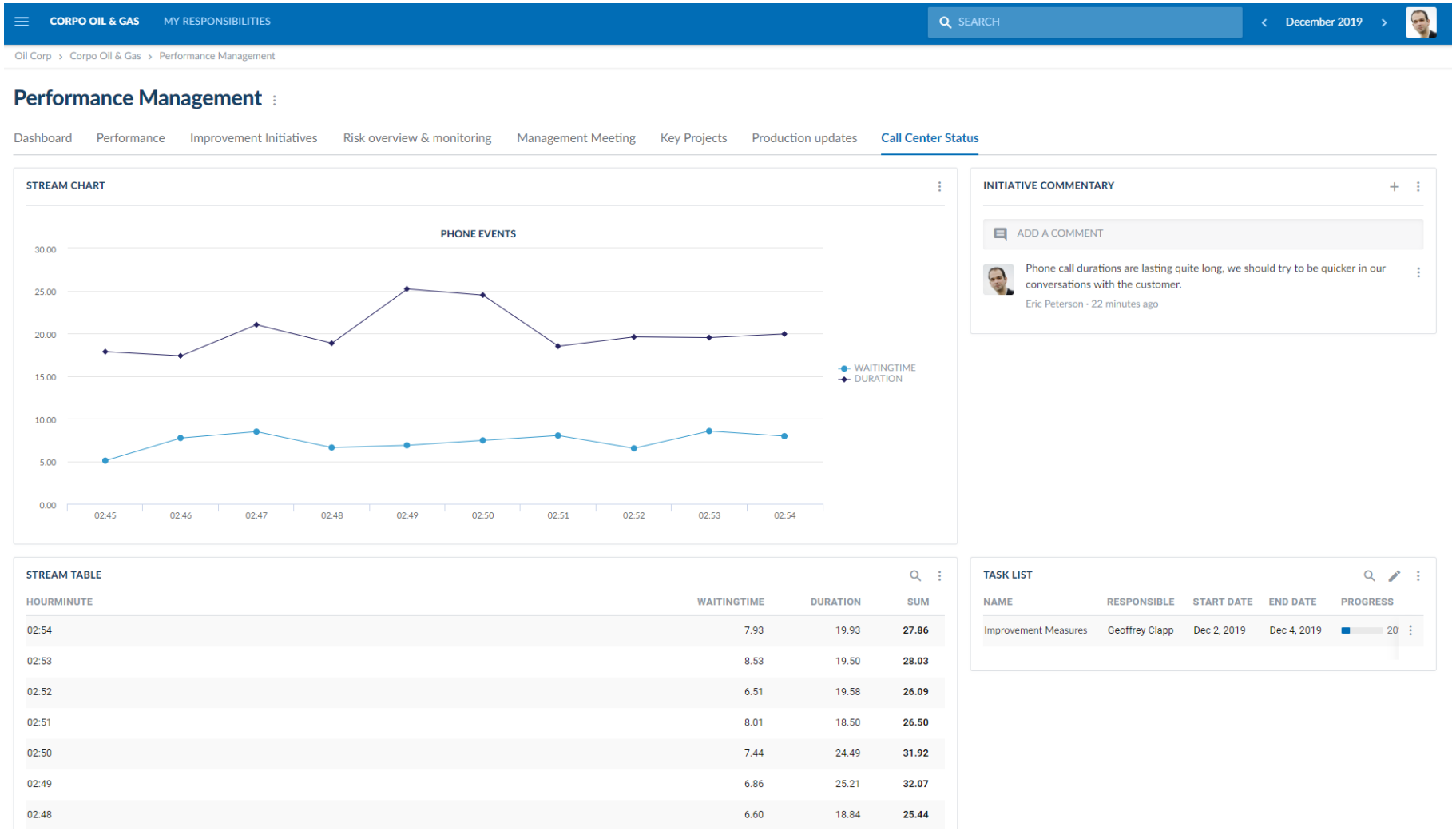


Figure 5.11: Second Result.

These figures illustrates a typical web interface in BMP. However, instead of having the chart and table connected to a batching source they connect to the data stream itself, which continuously updates the table. This can be shown by looking at the different timestamps in the two figures, where the newest value in Figure 5.10 is one of the oldest values in Figure 5.11. Furthermore, with these real-time illustrations together with the already existing features of BMP, allows the business experts to see quick changes in their systems and act upon them through other objects in BMP.

5.4 Analysis

The main goal of this prototype, is to provide an interface for business experts into data streams. Without the need for programming, these business experts would be able to configure these data streams. The resulting stream should then be displayed in a table or chart of the business expert's choosing. This goal was deduced through Corporater's requirements introduced in Chapter 1. In this section we will look into the prototype's fulfillment of this goal and analyze its limitations.

Based on the results in the previous section, it is clear that Business experts are able to get processed stream data into BMP. Furthermore, calculations could be set in Configuration Studio that is forwarded to Spark for execution. This is achieved by generalizing methods in Spark to such a degree that the configuration can be set in a different environment. Thus, the concept goal is proven by this prototype. However, sending configuration from BMP to Spark is not implemented thus far. The configurational examples shown in Section 5.2 are only how it could happen. In reality these configurations are set in a static class that Spark can request when needed in the calculational pipeline. This class is a separate component where all the calculational configuration in the pipeline exists. Thus, the concept of generalizing methods in Spark to the degree of setting configuration in a different environment still holds. This means that it is possible for Corporater to integrate with this system, and our proof still holds.

Limitations

A different subject in this analysis is the limitations of our prototype. From the evaluation chapter, it is known that Spark's latency is considerably higher than our other stream processors. Additionally, it has limited support towards inputs and outputs. These and other limitations were considered in the previous chapter, where Spark's benefits over-weighted these limitations. However, when developing this prototype there were more limitations of the system appearing. One of the most profound issues discovered is that 'Spark Analytics', which analyzes the calculational pipeline before execution, does not allow nested aggregations. This means that more complex combinational aggregations are not possible in the same pipeline. One solution to this problem could be to first do one aggregation then

output the result to a message receiver system. This system could then feed the output into Spark again through a different streaming instance that would do the second aggregation. Unfortunately, because of the time constraint of this thesis, this was not possible to test. Furthermore, this might not be the most optimal solution, but it is one solution that probably can enable nested aggregations.

Chapter 6

Conclusion & Future Directions

In this thesis, we wanted to investigate and develop a prototype that enabled business experts to utilize data streams without the need of a developer. With integrational support for Corporater's systems, this prototype should also perform adequately to their requirements and scenario.

We provided the investigation through an evaluation. From this evaluation we were able to recommend an architecture and stream processor that suffices Corporater's needs. Additionally, this evaluation provides a general overview of the leading architectures and stream processors differences. This allows for other businesses in a similar situation to draw beneficial value from its generality.

Based on the recommendation, we created a prototype. By implementing it in a generalized way, enabled compatibility with any type of GUI or API that are able to send configuration to the prototype. Further, we created an experiment utilizing our prototype. This created an end-to-end connection from the data source into Corporater's Business Management Platform (BMP). With this experiment we have proven that it is possible for business experts to visualize and configure data streams without the need of a developer.

Corporater has now been provided with crucial insight to the stream processing landscape. The research proves that today's stream processors have the integrational capacity to support Corporater's requirements, which enables future development of integrating stream processing support into their systems.

6.1 Future Directions

More Functionality and Generalization

There is definitely a need for more functionality in the system. For instance, utilizing a SQL parser to convert user configuration to SQL would enable less development in Spark. It would only be necessary to send the converted query directly to it and Spark could correctly execute this query. Furthermore, improving the generalization of the pivot method within Spark would provide a new feature few other systems can provide to their users.

Utilizing the machine learning library is also a possible extension to the system. By generalizing this library, would provide configuration and modifications of crucial real-rime predictions and forecasting to business users. This is also a rarity among business experts and could further give them an edge in a competitive market.

Kafka and Spark

An interesting software extension to Spark is Kafka. Surrounding Kafka around Spark can enable more security, fault tolerance, input and output support. With this implemented, it would be interesting to see whether Spark could run small batch jobs on the stream instead. This would enable all the functionality of the batch such as pivoting to be utilized on the stream. Additionally, this could also solve the limitation of nested aggregations, which would allow for more complex combinational aggregations to take place. Another interesting possibility with this setup, is to have some simple preprocessing done in Kafka. This would enable Spark to only focus on the most time consuming jobs and possibly increase the overall performance.

Spark Continuous Processing

Continuous processing is a new mode introduced in Spark 2.3 that branches away from the micro-batch approach. It is an experimental feature within the Structured Streaming library that enables Spark to achieve millisecond latency [37]. This is done through launching a set of long-running tasks that continuously read, process and write data. Furthermore, this feature enables Spark to compete with the other low-latency systems out there such as Flink, Kafka and possibly even Storm. Utilizing this feature within our prototype would enable more use cases for the system. The business users could then be even closer to real-time processing and possibly give them a larger edge in the market.

Apache Druid

Apache Druid is a high performing analytics database that specializes in features around event streams [38]. It is not considered a data warehouse. However, they are utilizing architectural ideas from the data warehouses such as column-oriented

storage. By creating a separate prototype utilizing Druid instead of Spark we would be able to evaluate their differences. Further, it would be interesting to see whether Druid is able to provide the same features and functionalities as Spark.

Apache Beam

Apache Beam is a system that provides a model which unifies both batch and stream data processing [39]. In our case it can be considered as an advanced Serving layer, that is able to send configurations down to one of their supporting pipeline runners (stream processors). Currently they are supporting Spark, Flink, Apex, Samza, Gearpump, and Google Cloud Dataflow. By switching out our H2 Serving layer, it would be interesting to see if the 'Business-In-Control' concept is more straightforward with this system compared to the current prototype.

List of Figures

Listings

List of Tables

Bibliography

[1] Microsoft. Azure Stream Analytics. https://azure.microsoft.com/en-us/services/stream-analytics/?&OCID=AID719817_SEM_BOZhH2AQ&lnkd=Google_Azure_Brand&dclid=CjgKEAjw0N3nBRCT2dvip9236lcSJABSb_usEOsk73Ot95GnlYrViQC24tLP2ah3I__QFCYhZVvA9vD_BwE, 2019.

[2] Amazon. Amazon Kinesis. https://aws.amazon.com/kinesis/, 2019.

[3] N. Mars & J. Warren. *Big Data: Principles and best practices of scalable real-time data systems*. Manning, 2015.

[4] Jay Kreps. Questioning the Lambda Architecture. https://www.oreilly.com/ideas/questioning-the-lambda-architecture, 2014.

[5] Milinda Pathirage. Kappa Architecture. http://milinda.pathirage.org/kappa-architecture.com/, 2014.

[6] Ampool. EMERGING DATA ARCHITECTURES – LAMBDA, KAPPA, AND BUTTERFLY. https://www.ampool-inc.com/emerging-data-architectures-lambda-kappa-and-butterfly/, 2016.

[7] S. Perera. A Gentle Introduction to Stream Processing. https://medium.com/stream-processing/what-is-stream-processing-1eadfca11b97, 2018.

[8] M. Dayarathna & S. Perera. Recent Advancements in Event Processing. https://www.researchgate.net/publication/323160411_Recent_Advancements_in_Event_Processing, 2018.

[9] S. Perera. From SQL to Streaming SQL in 10 Minutes. https://wso2.com/library/articles/2018/02/stream-processing-101-from-sql-to-streaming-sql-in-ten-minutes/, 2018.

[10] Apache Kafka. Kafka: A distributed streaming platform. https://kafka.apache.org/documentation/, 2019.

[11] Apache Storm. Kafka: A distributed streaming platform.
http://storm.apache.org/, 2019.

[12] Apache Flink. What is Apache Flink.
https://flink.apache.org/flink-architecture.html, 2019.

[13] Chandan Prakash. Spark Streaming vs Flink vs Storm vs Kafka Streams vs Samza : Choose Your Stream Processing Framework.
https://why-not-learn-something.blogspot.com/2018/03/spark-streaming-vs-flink-vs-storm-vs.html, 2018.

[14] Apache Spark. Apache Spark.
https://spark.apache.org/, 2019.

[15] Apache Samza. Apache Samza.
http://samza.apache.org/, 2019.

[16] Apex. Apache Apex.
https://apex.apache.org/, 2019.

[17] Google. CLOUD DATAFLOW.
https://cloud.google.com/dataflow/, 2019.

[18] V. Narasayya S.Chaudhuri, U. Dayal. An Overview of Business Intelligence Technology.
https://cacm.acm.org/magazines/2011/8/114953-an-overview-of-business-intelligence-technology/fulltext#F1, 2011.

[19] D. J. Power. A Brief History of Decision Support Systems.
http://dssresources.com/history/dsshistory.html, 2007.

[20] M. N. Frolick & T. R. Ariyachandra. Business Performance Management: One Thuth.
https://web.archive.org/web/20110719192210/http://snyfarvu.farmingdale.edu/~schoensr/bpm.pdf, 2011.

[21] Corporater. ACTION DRIVEN BUSINESS DASHBOARDS.
https://corporater.com/en/business-solutions/kpis-and-dashboards/, 2019.

[22] M. Rouse. operational intelligence (OI).
https://searchbusinessanalytics.techtarget.com/definition/operational-business-intelligence, 2018.

[23] GreenTree. Operational Intelligence.
http://www.appliedbusiness.co.uk/operational-intelligence-ebook/, 2016.

[24] Bobby Evans. Benchmarking Streaming Computation Engines at Yahoo!
https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at, 2015.

[25] Elkhan Shahverdi. Comparative Evaluation for the Performance of Big Stream Processing Systems.
https://www.semanticscholar.org/paper/Elkhan-Shahverdi-Comparative-Evaluation-for-the-of-Shahverdi-Sakr/4d2887513b538812809d5c7336978a5a189009d9, 2018.

[26] Burak Yavuz. Benchmarking Structured Streaming on Databricks Runtime Against State-of-the-Art Streaming Systems.
https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at, 2017.

[27] Aljoscha Krettek. The Curious Case of the Broken Benchmark: Revisiting Apache Flink vs. Databricks Runtime.
https://www.ververica.com/blog/curious-case-broken-benchmark-revisiting-apache-flink-vs-databricks-runtime, 2017.

[28] A. Katsifodimos R. Samarev H. Heiskanen V. Markl J. Karimov, T. Rabl. Benchmarking Distributed Stream Data Processing Systems .
https://www.researchgate.net/publication/323392536_Benchmarking_Distributed_Stream_Processing_Engines, 2018.

[29] C. Queiroz R. Buyya X. Zhao, S. Garg. A Taxonomy and Survey of Stream Processing Systems.
https://www.sciencedirect.com/science/article/pii/B9780128054673000119, 2017.

[30] Iman Samizadeh. A brief introduction to two data processing architectures — Lambda and Kappa for Big Data.
https://towardsdatascience.com/a-brief-introduction-to-two-data-processing-architectures-lambda-and-kappa-for-big-data-4f35c28005bb, 2018.

[31] Kafka. Kafka Streams.
https://kafka.apache.org/22/documentation/streams/, 2019.

[32] Storm. Stream API Overview.
http://storm.apache.org/releases/2.0.0-SNAPSHOT/Stream-API.html, 2019.

[33] Flink. Dataflow Programming Model.
https://ci.apache.org/projects/flink/flink-docs-release-1.8/concepts/programming-model.html, 2019.

[34] Spark. Structured Streaming Programming Guide.
https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html, 2019.

[35] Neha Narkhede. Apache Flink and Apache Kafka Streams: a comparison and guideline for users.

https://www.confluent.io/blog/apache-flink-apache-kafka-streams-comparison-guideline-users/, 2016.

[36] Storm. Documentation.
https://storm.apache.org/releases/2.0.0-SNAPSHOT/index.html, 2019.

[37] T. Das S. Zhu J. Torres, M. Armbrust. Introducing Low-latency Continuous Processing Mode in Structured Streaming in Apache Spark 2.3.
https://databricks.com/blog/2018/03/20/low-latency-continuous-processing-mode-in-structured-streaming-in-apache-spark-2-3-0.html, 2018.

[38] Druid. Apache Druid (incubating) is a high performance real-time analytics database.
http://druid.io/, 2019.

[39] Beam. Apache Beam Overview.
https://beam.apache.org/get-started/beam-overview/, 2019.

Appendix A

Github Repository

The link to relevant code and environment that is implemented for our thesis is provided in this link:
`https://github.com/nicolai-vs/datmas`