Scalability Benchmarking of Stream Processing Engines with Apache Beam

Bachelor’s Thesis

Jan Robert Bensien

March 28, 2021

KIEL UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
SOFTWARE ENGINEERING GROUP

Advised by: Prof. Dr. Wilhelm Hasselbring
Sören Henning, M.Sc.
Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 28. März 2021

iii
The rapid increase in data due to modern developments, such as Industry 4.0, Internet of Things, artificial intelligence or the analysis of user interaction on websites led to a plethora of new technologies dealing with the challenges associated with the four Vs of Big Data. One approach for dealing with these challenges, stream processing, gained special interest due to its ability to handle high loads while providing real-time results. Due to this, many different stream processing frameworks emerged, each with its own design decisions.

In this thesis, we will benchmark the horizontal scalability of two stream processing frameworks. To achieve this, we will apply the benchmarking method Theodolite and benchmark four already identified use cases. We will focus on Apache Beam as a programming API using Apache Samza and Apache Flink as stream processing backends. Additionally, we will adjust the framework of Theodolite to support the execution of Beam pipelines using Apache Flink and Apache Samza as stream processing engines. Following, we will take the necessary steps to execute our use cases in a private cloud environment, orchestrated by Kubernetes. We found that using Flink as processing backend is advantageous in three use cases, while using Samza as processing backend results in better results for one use case.
# Contents

3 Theodolite Benchmark Implementation in Beam 21
   3.1 Communication with Apache Kafka .......................... 21
      3.1.1 Naive Watermark Approximation ......................... 23
      3.1.2 Idle Advancement .................................. 24
   3.2 UC1: Database Storage .................................. 25
   3.3 UC2: Downsampling ..................................... 26
   3.4 UC3: Aggregation based on Time Attributes ................. 27
   3.5 UC4: Hierarchical Aggregation .......................... 29

4 Embedding the Use Cases into Theodolite 33
   4.1 Adjusting Theodolite to support the Execution of Beam Pipelines using the Flink and Samza Runners ................................. 33
   4.2 Deployment of Beam Pipelines in Kubernetes using the Samza Runner ................................ 35
   4.3 Deployment of Beam Pipelines in Kubernetes using the Flink Runner ................................ 36

5 Evaluation 39
   5.1 Experiment Setup ......................................... 39
   5.2 Configurations for Experiments ............................. 39
   5.3 Preparatory Experiments ................................ 40
      5.3.1 Number of Partitions ................................ 40
      5.3.2 The Impact of Beam’s Metrics on Scalability ........... 41
      5.3.3 The Impact of fasterCopy and Beam’s Metrics on the Scalability of the Flink Runner ......................... 42
   5.4 Results and Discussion ................................ 44
      5.4.1 UC1: Database Storage ................................ 44
      5.4.2 UC2: Downsampling .................................. 45
      5.4.3 UC3: Aggregation based on Time Attributes .......... 46
      5.4.4 UC4: Hierarchical Aggregation ......................... 47
   5.5 Threats to Validity ......................................... 48

6 Related Work 49

7 Conclusions and Future Work 53
   7.1 Conclusions ................................................. 53
   7.2 Future Work ................................................. 54

Bibliography 57
Chapter 1

Introduction

1.1 Motivation

Modern software systems face the challenge of processing steadily increasing amount of data [Chen et al. 2014]. Data processing systems can be divided into batch systems and stream processing systems. Batch processing systems process chunks of data as a whole, while stream processing systems process the data on arrival instead of segmenting it into batches which are processed with a possible delay. Stream processing frameworks gained popularity due to their inherent scalability [Carbone et al. 2020].

To support horizontal scalability, modern software systems are often structured as microservices [Hasselbring and Steinacker 2017; Hasselbring 2016; Villamizar et al. 2015; Alshuqayran et al. 2016]. These microservices often implement a range of common streaming use cases [Henning and Hasselbring 2021b]. Structuring system architectures through microservices allows to distribute the workload in elastic cloud environments [Hashem et al. 2015]. In this kind of environment, each microservice can be deployed as a logically separated unit with minimal dependencies to other microservices. These units can be deployed a variable amount of times to increase the processing resources of the application. Stream processing engines such as Apache Kafka Streams [Sax et al. 2018], Apache Samza [Noghabi et al. 2017], and Apache Flink [Carbone et al. 2015b] are used to implement these microservices. In practice, a developer will want to find the most suitable technology for a given use case, which is done by benchmarking said system. During that process, the performance of multiple systems combined with various workloads is measured and compared.

In this thesis, we plan to implement common use cases using Apache Beam [Apache Software Foundation 2020a] and apply the benchmarks with different stream processing backends to gain insight into the horizontal scalability of the stream processing engines. Apache Beam provides an API to implement stream processing applications which can be run using different stream processing frameworks as backends. As stream processing as a technology to process big data is highly dependent on being scalable, it raises the question if the more adaptive approach of Beam outweighs the possible overhead introduced with another abstraction layer. Choosing Beam for our implementation, it allows us to use the same implementation for every framework. This way, our results are not subject to differences which might occur when using different APIs for the implementation of the
1. Introduction

same use case and we can focus on comparing the scalability of the stream processing backend instead of optimizing different implementations.

1.2 Goals

1.2.1 G1: Implementing Common Use Cases of Stream Processing Engines in Apache Beam

In order to be able to gain representable benchmarking results, a range of common use cases will be implemented. We will adopt the four use cases which have already been identified in the work on the benchmarking framework Theodolite [Henning and Hasselbring 2021b]. These use cases are derived from the context of analyzing the power consumption in manufacturing enterprises. Reference implementations using Kafka Streams and Apache Flink are already provided for Theodolite. We will add to them and provide implementations using Apache Beam. This includes that we will provide a implementation of the Hierarchical Aggregation [Henning and Hasselbring 2020a] using the Dataflow Model. Using Apache Beam will allow us to execute the implementations with different stream processing frameworks as backend. This will be the basis for the following benchmarking experiments of different stream processing engines.

1.2.2 G2: Embedding the Use Cases into Theodolite

After achieving G1: Implementing Common Use Cases of Stream Processing Engines in Apache Beam, we can embed our implementation into Theodolite. To obtain this goal, we will adjust the execution of benchmarks in the scalability benchmarking framework Theodolite. We will provide the means to execute Beam applications using Apache Flink and Apache Samza as processing backends. This requires us to containerize our use cases implementations and create the correct configurations for the different stream processing engines. To containerize our use cases, we will use Docker as a containerization technology. Furthermore, we need to configure each stream processing engine manually.

1.2.3 G3: Execution of Benchmarks and Evaluation of Scalability

Lastly, we will execute a range of benchmarks of different stream processing engines according to Theodolite’s scalability benchmarking method using our use case implementations. This includes different configurations for the same stream processing engine to find the most suitable configuration, however the focus is set on the comparison of different stream processing engines using comparable configurations. Furthermore, we evaluate the scalability of Beam as another abstraction layer.
1.3 Document Structure

This thesis is structured as follows: Chapter 2 introduces the foundations of stream processing and the benchmarking method to be applied. The benchmarking method includes four use cases which we base our implementation on. Furthermore, it discusses the technologies which we use to execute our experiments. In Chapter 3, we discuss our implementation of the earlier proposed use cases using Apache Beam. Following, in Chapter 4, we take a deeper look at Theodolite and how we added support for the used stream processing engines. This includes the deployment of the applications in a cloud environment. In Chapter 5, we will execute and evaluate experiments on the horizontal scalability of Apache Samza and Apache Flink in combination with Beam. We will discuss related work in Chapter 6. Chapter 7 summarizes this work and looks at possible future work.
Foundations and Technologies

In this chapter, we will explain the foundations of our thesis and introduce the technologies we used. First, we will introduce the concept of distributed stream processing. One model that applies distributed stream processing is the Dataflow Model which is implemented as the framework Apache Beam, as will be described in detail. While Beam provides the necessary model to describe a stream processing application, multiple different stream processing frameworks can be used as the processing backends to execute said applications. We will take a closer look at Apache Flink and Apache Samza as processing backends. We will furthermore characterise how we define the scalability of stream processing frameworks and measure it. This was adopted from the benchmarking framework Theodolite [Henning and Hasselbring 2021b], which we will also cover in this chapter. Furthermore, we will introduce the necessary technologies we used to execute the benchmarking experiments.

2.1 Distributed Stream Processing

The shift to increasing amounts of data forced the research community to develop new ways to process it. Reasons for the increasing data amounts include Industry 4.0, the Internet of Things, the analysis of user data from web applications, such as online shops or social media, and the training of neural networks. The change in the characteristics of the data, for example, the faster decay of data [Wingerath et al. 2016] or the sheer increase in volume [Chen et al. 2014], and the need for real-time results in many application domains, such as network monitoring and fraud detection [Stonebraker et al. 2005], lead to new technologies being developed to cope with these challenges. Processing of data using databases which store the content on the disk and later use batch queries to process them got more and more impractical. Nowadays, real-time processing of data which is too big to fit into memory while processing is wanted [Wingerath et al. 2016]. Stream processing might be the solution for processing big data in near real time.

In contrast to batch data, where the full data is available before processing and hence bounded, data streams are possibly unbounded and often consists of data which is generated continuously over time, making it not fully available before processing begins. It is however also possible to represent batch data as a special case of bounded streams [Carbone et al. 2015b]. Stream processing systems are not able to control the data arrival rate and order of data. Furthermore, they are not able to determine if it is possible to store
2. Foundations and Technologies

the full data while processing. Thus, data elements have to be processed as soon as they arrive using limited memory. The data can be either produced by sensors, user interaction, other stream processing systems or many other sources. It is also possible to feed databases into stream processing systems. This can be done using a changelog to process updates at real time or consuming snapshots.

Often times, the data source will not continue to persist the data after feeding it into the stream processing system. This leads to the problem that, upon failure, the stream processing engine can not reconsume the data to restore the state and needs to adopt different strategies to gain fault tolerance. Requirements for a stream processing engine, in the following referred to as SPE, are low latency and high throughput. Furthermore, an SPE needs to produce correct results even for out-of-order data or delayed data. To achieve this, it needs to reason about progress and completeness of the computation [Akidau et al. 2015]. Modern stream processing use cases often have the characteristic that the data input rate is highly fluctuating, which leads to the requirement, that the SPE has to be able to adapt to different workloads without sacrificing performance.

Research in the field of stream processing has been active since at-least the 1960s [Stephens 1997]. There has been especially high interest since the late 1990s and early 2000s. Early ideas have been implemented for example in TelegraphCQ [Chandrasekaran et al. 2003] as a way to express and compute continuous queries. In the early 2000s, challenges such as sliding window aggregations, fault-tolerance and load balancing emerged. Based on processing ordered event streams, systems such as IBM System S, Esper, and ORACLE CQL/CEP have been developed between 2000 and 2003 [Fragkoulis et al. 2020]. The introduction of Google’s MapReduce [Dean and Ghemawat 2008] and the shift towards cloud computing sparked the interest in scalable parallel processing using distributed clusters. Ideas such as out-of-order processing, state management and processing guarantees got into the focus of development. Stream processing frameworks such as Apache Spark [Zaharia et al. 2016] and Apache Flink [Carbone et al. 2015b] lay their focus on these ideas.

Nowadays, the most common notation of stream processing applications is that of a directed acyclic graph [Stephens 1997], as seen in Figure 2.1, where the nodes represent data transformation steps, data sinks, or sources, and the edges represent streams between single steps.

In the following, we describe some of the core concepts that need to be addressed in the design of modern stream processing frameworks.

2.1.1 Time in Stream Processing

Real-time processing using stream processing engines, forces the developer to reason about how time has to be handled in a given scenario. The most common used time notions are event time and processing time. Processing time is the time at which the data is being processed at any step in the SPE. This is based on the clock of the underlying hardware. Event time is the time at which the event occurred at the source. Event time is based on
2.1. Distributed Stream Processing

![Stream processing application](image)

Figure 2.1. Stream processing application

Network latency and the time at which the data is generated may introduce out-of-order or delayed data. We define out-of-order events as events which arrive in an SPE after events with a later timestamp. To detect and handle out-of-order data, the SPE needs to track the progress of its processing. The process should reflect the time of the oldest unprocessed event. As we have no full knowledge of the data source, it is only possible to estimate the progress. To track progress and cope with out-of-order data, mechanisms such as slack, heartbeats, and low-watermarks are used [Fragkoulis et al. 2020].

Using **slack**, the system waits for a fixed grace period of time for out-of-order events. **Heartbeats** are emitted by the data source and carry information on how far the stream has already been processed. A **heartbeat** with timestamp \( t \) carries the information that every event before \( t \) has been processed. **Heartbeats** can also be calculated by observing network metrics. A **low watermark** for an attribute is the lowest value of said attribute in a given subset of the stream. Using a **low watermark** for the progress allows to probabilistically estimate how far the stream has been processed. In contrary to **heartbeats**, the **low watermark** has to be estimated within the SPE using the timestamps of the already processed events or the current clock time. Most real world scenarios do not allow establishing a perfect **watermark** estimation, since the knowledge within the system is insufficient. Based on this, **watermarks** can not guarantee to reflect the true progress, but allow to make a best effort guess.

### 2.1.2 State and Stateful Functions

Every side effect while processing produces some kind of state. State has to be maintained for active windows and aggregates. Especially continuously running applications with possibly hundreds of instances needs a sophisticated mechanism for state management [Anis Uddin Nasir 2016]. This needs to address challenges encountered when scaling the
2. Foundations and Technologies

number of processing units and partitioning the processing steps. Recreating state upon failure is another non-trivial challenge. As the SPE may not be able to reprocess the input data, either because the source does not maintain it after ingestion, or because it is simply too time-consuming, mechanism such as state checkpointing and changelogs are used. In the programming model, state can be either explicitly declared and managed or implicit [Fragkoulis et al. 2020]. Modern sophisticated SPE often use a hybrid method, wherein, for example, state for window aggregations is implicitly managed while also allowing the programmer to explicitly declare state cells or stores. Most SPE, such as Samza, Flink, and Kafka Streams use a state which is managed by the system itself and declared by the user [Carbone et al. 2017; Noghabi et al. 2017].

Key-level state, where logical partitioning is supported by defining keys for each chunk of state, are the most common type of state used by modern SPE [Fragkoulis et al. 2020]. The counterpart to key-level state is the task-level state. Task-level state partition the state according to every computation task. This allows only one instance per task and reduces the parallel processing ability of an SPE as the state does not scale as parallel as the data. The advantage of task-level state lays in computing global aggregates using static state sizes.

2.1.3 Fault Tolerance

Most modern SPE can execute complex applications resulting in arbitrary long-running applications while supporting different mechanisms to gain fault tolerance. Fault tolerance is the ability of a system to resume its execution upon failure, while also producing correct results. Without mechanisms for gaining fault tolerance, it would not be possible to restore the current state upon failure, as the processed stream may no longer be available. Because of the distributed and highly scalable nature of many SPE, they may be even especially prone for faults.

Supported processing guarantees by modern SPE are mostly exactly-once or at-least-once processing guarantees. At-least-once refers to when a system produces correct results after failure, with the exemption of possible duplicates. Exactly-once processing guarantee on output refers to when a system, even in case of failure, always produces the same output. However, instead of choosing to benchmark the system’s ability to recover from faults, we chose to benchmark the system’s ability to scale according to different workloads under the assumption of no failures.

2.2 The Dataflow Model for Stream Processing

In 2004, Google presented the MapReduce model [Dean and Ghemawat 2008]. Based on this, Google continued internal development of big data solutions. Meanwhile, there has been a growing open source community, which developed its own big data solutions. The resulting frameworks, such as Hadoop, implemented models which have been influenced
by MapReduce. While the Google internal solutions where homogeneous in file formats and used programming languages, the open source solutions covered a wide array of different programming languages and use cases.

Eventually, Google published the programming model Google Dataflow [Akidau et al. 2015]. The main idea behind the Dataflow Model is to provide a unifying model for stream and batch processing. With this model, it is possible to implement applications using many common advantages of SPE’s, such as event-time, unbounded unordered data sources, processing guarantees and low latency. The Dataflow Model does not distinguish between bounded and unbounded data sources consumed as batch, micro-batch, or stream. This distinction is solely made when looking at the execution engine and does not change the application implementation itself. The Dataflow Model is not a stream processing engine, but a programming model for stream processing applications. In the following, we will introduce some of the core concepts of the Dataflow Model. We will not consider execution engine specific aspects, such as state management, correctness, and fault tolerance. They are the basis for the decision which engine to use, but not how to implement an application using the Dataflow Model. Event time and processing time, as addressed in Section 2.1, are both considered in the Dataflow Model.

The first implementation of the Dataflow Model was provided for running stream processing applications on Google’s Cloud Dataflow service\(^1\). Later on Apache Beam was built upon this first implementation and aimed to support a multitude of different processing backends. In the following, we will take a look at the semantics of the Dataflow Model. These concepts are also used in Beam.

### 2.2.1 Pipelines, PTransforms, and PCollections

Often, stream processing applications are defined as graphs. In the Dataflow Model, the **Pipeline** is the core abstraction of the application. It represents the graph and consists of **PTransforms**, as edges, applied to **PCollection**, as nodes. **PCollections** are unordered bags of elements. Depending on the input source, a **PCollection** can either be bounded or unbounded. Furthermore, every element in a PCollection has a timestamp attached and is assigned to a window. Without applying non-default windowing strategies, as described in Section 2.2.3, a global window is used, which includes the whole stream. **PTransforms** are operations which are applied to a **PCollection**. The processing logic is provided by a function object and applied to every element in the PCollection. This results in a new **PCollection** and corresponds to the edge in a graph, as seen in Figure 2.1.

### 2.2.2 ParDo and GroupByKey

The core transformations of the Dataflow Model, which can be applied on key-value pairs, are **ParDo** and **GroupByKey**. **ParDo** is the most generic parallel processing transformation,
2. Foundations and Technologies

which applies a user-defined function, a DoFn, on every input element, resulting in an arbitrary number of output elements per input element. ParDo can be compared to the map step in MapReduce. GroupByKey selects all data in a PCollection for each key. Which is comparable to a map from each distinct key to a group of associated values or the shuffle step in MapReduce. Using only the core transformations, a reduce step can be implemented as a GroupByKey transform followed by a ParDo with a single output for a group of data for a given key.

2.2.3 Windowing

Often an application will compute statistics, such as averages of energy consumption, over well-defined periods of time. To divide the input stream, based on for example, the time domain, into chunks representing a fixed period, windowing strategies are applied. In the Dataflow Model, windows based on a time domain are considered. It is argued, that even tuple-based windowing is based on a logical time domain using successively increasing logical timestamps [Akidau et al. 2015]. Three different window types are distinguished: fixed windows, sliding windows, and session windows.

Fixed windows have a constant window size, which defines the time period between the windows boundaries, for example, daily windows which capture the elements of a single day. Sliding windows have, additionally to a window size, a slide period. The slide period defines at which point in time we open a new window, for example, windows which are created every day and capture 7 days. Whenever the slide period is smaller than the window size, the windows will start overlapping. It is also possible to define fixed windows as sliding windows with the same slide period and window size. Session windows are defined by a timeout gap. Events that occur within the timeout gap are grouped together in a window.

Windowing is broken into two steps: assigning the element to arbitrarily many windows and merging windows to create data driven windows. In the assignment step, a new copy of the key-value pair together with the timestamp and corresponding window is created for every window that the event is assigned to.

2.2.4 Triggers

Triggers are used to define when to emit the results of a window. Watermarks, as addressed in Section 2.1, are not sufficient in order to decide whether the computation of a window is finished. Data might be arriving after we pass the watermark or the watermark might progress too slow because of single events, which can hold back the advancement of the watermark. This can lead to wrong results or higher latency.

Triggers are used to complement the use of watermarks, as they allow to specify the behaviour of the window in more detail. They allow to provide low-latency best effort results while also ensuring eventual consistency. This is only possible when multiple results, so-called panes, for a window are allowed. Triggers can be either based on event time,
processing time or can be data driven. Furthermore, the behavior upon triggering a pane can be discarding or accumulating. When discarding is chosen, the system will discard the window content upon triggering. When accumulating is chosen, it allows the window contents to be used for later refinements. Accumulating is used, when the downstream step will overwrite earlier results with newly triggered ones. This mimics the behavior of the Lambda architecture [Marz and Warren 2013], as low latency results will be overwritten by more refined results.

The relation of triggers to windows can be summarized such that windowing determines where in event time data are grouped together for processing, while triggering determines when in processing time the results of groupings are emitted as panes [Akidau et al. 2015].

### 2.3 The Unified Programming Model Apache Beam

After the publishing of the Dataflow Model, Google donated the programming model to the Apache Foundation as the project Apache Beam. Beam can be seen as an implementation of the Dataflow Model. It supports the full advantages of the Dataflow Model with focus on the flexibility and ease of development. It also tries to build the bridge between different stream processing technologies through the unified programming model. On top of this
2. Foundations and Technologies

Beam adds the ability to use different runners to execute the Dataflow implementations. Runners are used to execute the application and provide the stream processing backend. The runner will take the constructed pipeline and translate it into code corresponding to the API of the chosen framework. Furthermore, it will make all necessary steps to start the execution. The focus is not set on providing another backend to run stream and batch processing jobs, but on providing a unified model for different backends.

An overview of the process of implementing and executing an application with Beam is shown in Figure 2.2. At the top level, the different SDKs of Beam are shown. Using these SDKs, a pipeline can be implemented and in the next step, a backend for the processing can be chosen. The pipeline will then be executed by the provided runner for this backend.

Beam aims to support different categories of users. Firstly, the end user, who can use Beam’s flexibility to choose the SDK and the stream processing backend based on their needs and expertise. Beam also supports SDK writers. The aim is to allow other open source projects to make the Beam execution backends available in other programming languages and even narrow domain specific languages. Lastly, other stream processing frameworks have the possibility to implement a Beam runner to gain access to different SDKs and the programming model of Beam. For example, Samza chose not to support event time using their native APIs, but implemented a Beam runner, which allows to use event time using Beam’s programming model.

We chose Beam to implement our benchmarks in order to be able to execute them using different stream processing backends. As SDK we chose Java since Java is already used to implement the benchmarks in native Flink and Kafka Streams. As runners, we chose Samza and Flink. We furthermore might gain insight into how the added abstraction layer impacts the performance of the stream processing framework.

2.4 Stream Processing Frameworks

In the following, we will showcase the stream processing frameworks which we have chosen as execution backends for our experiments. We chose modern stream processing frameworks, to reflect the current state of development and gain comparability. As two of our use cases use event time, we only chose frameworks which support event time. We do however have to highlight that Samza currently only supports event time using the Beam API and does not support event time using their native APIs.

We chose the runners for Apache Flink and Apache Samza to execute our applications. Apache Flink [Carbone et al. 2015b] is an open-source stream processing framework which is, as well as Samza and Beam, organized within the Apache Foundation. When introducing Flink, the focus laid on its ability to combine batch processing with stream processing while using the same API. Batch processing is seen as a sub category of stream processing, where the source is bounded and the order of the single data elements is not of relevance. Flink’s different APIs allow the programmer to implement different use cases with ease.
2.4. Stream Processing Frameworks

Notable are the DataSet API with focus on batch processing and the DataStream API with focus on stream processing. Furthermore, APIs for more specific use cases such as machine learning, graph processing and the table API are provided. Flink is used in production by many companies such as Alibaba, Amazon Kinesis Data Analytics, and Ebay.

Apache Samza [Noghabi et al. 2017] is a stream processing framework with focus on the performance of stateful processing. Samza allows to process infinite streams of messages. Furthermore, similar to Flink, it supports the processing of finite datastreams from streams, database snapshots, and file systems. Natively, Samza offers APIs that allows the same application logic for both stream processing and batch processing. A high level streams API, which is for further ease of development based on the widely adopted Java 8 Stream package, and a low level task API are provided. Furthermore, an SQL API provides a declarative query language for the implementation of stream processing applications. Samza is widely adopted by top-tier companies (for example, LinkedIn, Uber, Netflix) [Noghabi et al. 2017]. As of 2017, LinkedIn uses Samza in combination with Apache Kafka as messaging framework in hundreds of application with more than 10,000 containers [Noghabi et al. 2017].

Both Samza and Flink aim to provide high throughput and low latency processing of continuous data streams. In the following, we will take a closer look at some of the key features of both stream processing engines, which are shown in Table 2.1.

<table>
<thead>
<tr>
<th>Time notions</th>
<th>Apache Samza</th>
<th>Apache Flink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out-of-order data</td>
<td>Low-Watermark</td>
<td>Low-Watermark</td>
</tr>
<tr>
<td>Progress tracking</td>
<td>Timestamp</td>
<td>Timestamp</td>
</tr>
<tr>
<td>State management</td>
<td>In-Memory, File, RocksDB</td>
<td>In-Memory, RocksDB</td>
</tr>
<tr>
<td>Processing guarantees</td>
<td>at-least once</td>
<td>exactly once</td>
</tr>
<tr>
<td>Recovery data</td>
<td>local</td>
<td>remote</td>
</tr>
</tbody>
</table>

2.4.1 Tasks and Jobs

Samza and Flink both break down stream processing application graphs into subgraphs which are processed in parallel. These subgraphs are called tasks, and the whole application graph is referred to as the job. Both Flink and Samza clusters are used to run tasks in parallel on different nodes in so-called containers. Each task will be assigned a range of input partitions. To coordinate the containers, a central coordination agent is used. In Flink, the JobManager is used as the coordinator, while Samza uses one of the containers itself as coordinator.
2. Foundations and Technologies

2.4.2 Stateful Processing and Processing Guarantees

Both Flink and Samza provide the ability to execute stateful stream processing applications. For applications with smaller state, an in-memory state can be used. Through support of a keyed state store on disk, they furthermore support states which would not fit in memory. For both SPEs, RocksDB\(^2\) can be used for this state store. RocksDB is a high-performance and low-latency single machine storage which supports layered caching. A least-recently-used algorithm is supported to keep the most accessed data in memory. As data from RocksDB needs to be serialized and deserialized, Samza adds another cache layer, where data is kept deserialized in memory. Each task gets associated with its own instance of a keyed state store, which stores data corresponding to each partition of the task. Flink and Samza support both local state, where the data is kept on the same node as the task, and external state.

2.4.3 Fault Tolerance

To gain fault tolerance, Samza stores a changelog of the local state of a task. This changelog is an append-only data structure which only captures incremental changes rather than a snapshot of the state. To enable fast replaying of the changelog, it is stored using Kafka. The size of the changelog is reduced by using Kafka’s compaction feature. Compaction is used to only retain the latest value for each key. Using this changelog, Samza achieves at-least-once processing. It is argued that at-least-once processing guarantees are sufficient for most use cases and it is possible to achieve exactly-once guarantees via implementation using the local state. Using this changelog, an overhead of only 5% was achieved [Noghabi et al. 2017]. One drawback of using a changelog is the increased overhead when restarting a failed container. Samza tries to reduce this overhead by placing the restarted container on the same physical unit as the failed container, the so-called host affinity. Using host affinity, a container can simply access the file containing the state on the host when restarted instead of replaying the changelog.

Flink uses a different approach to achieve fault tolerance. In Flink, a combination of stream replays and checkpointing is used. The mechanism used is the Asynchronous Barrier Snapshotting algorithm [Carbone et al. 2015a], which is inspired by the Chandy-Lamport algorithm [Chandy and Lamport 1985]. Flink uses barriers, which are inserted into the stream, to logically separate the stream into epochs. All records that flow before the barrier will be assigned to the snapshot with the barrier id as identifier. Once the barrier arrives at a sink, an acknowledgement, including the offset when the snapshot was started and a pointer to the stored state, will be sent to the checkpoint coordinator. When the checkpoint coordinator has received acknowledgements for all sinks, it has gained a complete snapshot of the state of all operators. Upon failure, Flink selects the last acknowledged checkpoint. The state of this checkpoint will be restored and the stream can start processing at the attached offset. Using this method, Flink achieves exactly-once processing guarantees.

\(^2\)https://rocksdb.org/
2.5. The Theodolite Method for Scalability Benchmarking of Stream Processing Engines

One important thing to note is that while the recovery data in Samza is stored locally and host affinity is used upon failure, Flink stores the recovery data externally. Samza’s host affinity does not guarantee that the task is started on the same node but LinkedIn’s internal tests [Noghabi et al. 2017] found that in most cases, host affinity did ensure that the recovery data was kept local. This might allow for faster recovery in case of failure.

2.4.4 Time Notions

Both Frameworks offer processing time and event time. Flink furthermore offers ingestion time, which is the point in time at which the record is first seen by the processing system. Samza currently only offers event time when using Beam as API.

In both Samza and Flink, the mechanism of low-watermarks as described in Section 2.1 are used to track the progress of the application.

2.5 The Theodolite Method for Scalability Benchmarking of Stream Processing Engines

Theodolite [Henning and Hasselbring 2021b] is the first benchmarking method for the scalability of stream processing engines in a use-case-oriented fashion. A benchmark is used in empirical computer science as a “standard tool for the competitive evaluation and comparison of competing systems or components according to specific characteristics, such as performance, dependability, or security” [v. Kistowski et al. 2015]. Theodolite’s benchmarks are designed as application-driven benchmarks. As such, a common use case is implemented as a microservice and deployed accordingly often to obtain the desired workload instance ratio. The system under test, which consists of microservices that performs stream processing, is observed in a realistic deployment including a messaging system, such as Apache Kafka, and a distributed cloud environment.

2.5.1 Benchmarks

Henning and Hasselbring [2021b] proposed four benchmarks which are derived from use cases identified in an Industrial Internet of Things project [Henning and Hasselbring 2021a]. These use cases aim to cover a wide range of modern application problems. As such, it is assumed that benchmarking them enables to transfer the findings to other application domains. In the following, we will specify the use cases and their original topology. All use cases use a messaging system, such as Apache Kafka, as data sources and sinks. The messages from the data sources are in the form of a key-value pair where the key is a identifier String and the value is the associated measurement data.
2. Foundations and Technologies

![Figure 2.3. Database storage [Henning and Hasselbring 2021b]](image)

UC1: Database Storage  The first use case is depicted in Figure 2.3. Events are provided by an input stream and a function is mapped onto them. Afterwards, the data is stored permanently in a database. This could be a simple filtering based on an ID or altering the events through a simple function.

UC2: Downsampling  In the use case Downsampling, as seen in Figure 2.6, the amount of incoming messages is reduced by grouping it into logical subsets and applying an aggregation, such as average, minimum, or maximum functions. This is the minimal example for every use case which aggregates over an incoming stream. For this, the incoming data is partitioned using a tumbling window [Carbone et al. 2019]. In the next step, the corresponding data is aggregated, using a function such as sum, count, minimum, maximum, average, or population variance. As such, the data from the output stream is in a format fitted for the use of machine learning methods or data visualization.

UC3: Aggregation based on Time Attributes  In this use case, as opposed to Downsampling a time attribute is extracted and used for the partitioning of the incoming data stream. This enables the aggregation of, for example, all data that has been obtained on a Monday in the last month to detect seasonal patterns. In this use case, hopping windows [Carbone et al. 2019] are used, contrary to tumbling windows, which are needed in Downsampling. For example, a window duration of a month and an advance period of a day could be used. This way only relevant data of the last month are used for the aggregation and older data is discarded. This leads to multiple overlapping windows, which in return require multiple
2.5. The Theodolite Method for Scalability Benchmarking of Stream Processing Engines

Figure 2.5. Aggregation based on time attributes [Henning and Hasselbring 2021b]

Figure 2.6. Downsampling [Henning and Hasselbring 2021b]

aggregations to be performed.

UC4: Hierarchical Aggregation This use case is derived from analyzing sensor data, where often aggregated groups of sensor data are of importance [Henning and Hasselbring 2020a]. An example application for this could be monitoring the power consumption of different machine types in an industrial facility. Using Hierarchical Aggregation allows to group sensor data according to a configurable hierarchy. One stream provides sensor data and another stream tracks the changes to the hierarchy of the sensors. In the dataflow, the streams are joined, the according measurements are duplicated then aggregated according to their groups and lastly, for further processing, written to an output stream. The output stream also functions as input for the purpose of computing aggregations of subgroups which rely on earlier results. This dataflow is shown in Figure 2.4 and described in further detail in [Henning and Hasselbring 2020a].

2.5.2 Measuring Scalability with the Demand Metric

Herbst et al. [2013] defined scalability as the ability of a system to sustain increasing workloads by making use of additional resources. Furthermore, a system can be considered scalable if it meets certain quality criteria even in demanding situations [Weber et al. 2014]. Weber et al. [2014] outlined three attributes, which characterise scalability.

Henning and Hasselbring [2021c] applied these attributes to the context of stream processing and uses the following attributes for Theodolite. The input variable defines the load intensity that a system is exposed to. In stream processing load has the form of messages which are provided by a centralized messaging system. The system is using its provisioned resources, which may be increased, to handle the given load. To measure the scalability of a system when exposed to a load, defined by the load intensity and using its provisioned resources, a quality criterion is used. The quality criteria can be a service level objective (SLO), which sets a lower bound for the acceptable service level of the application. This is often the requirement, that all messages are processed in time.
Scalability can be further divided into vertical and horizontal scalability. A system is vertically scaled when the provisioned resources of existing nodes are changed, while horizontal scaling is the process of adding or removing nodes to a cluster [Weber et al. 2014]. In stream processing, the underlying hardware is often hidden behind an abstraction layer, such as the containerization technology Docker, which makes horizontal scaling the preferred method.

Henning and Hasselbring [2021c] proposed the resource demand metric for the scalability of SPEs. This metric is a function that maps load intensities to the resources, which are at least needed to satisfy a given SLO. Using the resource demand metric we can examine if a systems demand for resources scales linear, exponential or if there exists an upper bound for the processable loads.

The lag trend, is the SLO, which is the basis of the computation of the resource demand metric within Theodolite [Henning and Hasselbring 2020b]. Lag in a stream processing application describes how many messages are queued in the messaging system and await processing. When monitoring the system for extended periods of time, a trend can be computed by linear regression and the slope of this trend is used as the lag trend. The SLO, as a lower bound of the acceptable service level, is obtained by setting a maximum threshold for the slope. In our experiment we will use the demand metric to evaluate the horizontal scalability of Apache Flink and Apache Samza in combination with Apache Beam.

### 2.5.3 The Benchmarking Execution Framework Theodolite

To execute the benchmarks, we use the benchmarking execution framework Theodolite proposed by Henning and Hasselbring [2021b]. The benchmarking execution framework allows running subexperiments for different workloads and numbers of instances. Figure 2.7 shows Theodolite’s software architecture. Central to the framework is the experiment control. It configures the workload generator, starts, and replicates the instances of the stream processing engine, and is the component which starts each subexperiment. The workload generator is used to generate a workload of a chosen dimension. The dimension of workload has to be chosen according to the tested use case. The communication between workload generator and stream processing engine is handled by the messaging system. Currently, Apache Kafka as introduced in Section 2.6 is used as messaging system. The use case is implemented using a stream processing engine, in our case Samza and Flink in combination with Beam, and is the component which is being benchmarked. This component also feeds data to the monitoring component. On the one hand, the monitoring component enables an offline analysis, as it saves the monitoring data for later analysis. On the other hand, it allows the dashboard to visualize runtime information and verify that the configuration setup is as desired.
2.6 The Messaging System Apache Kafka

Apache Kafka [Kreps et al. 2011] is a messaging framework which allows defining streams of messages. These messages consist of key-value pairs. Producers can publish messages to certain topics. Topics are a data structure which in return can be subscribed to by a consumer. This way, topics act as a data store which publishes a continuous stream of messages. A consumer client can asynchronously consume from a variable amount of subscribed topics. In a consumer application, the consumer first subscribes to a topic which he realizes by creating one or more message streams for the needed topics. Using the provided iterator interface, it is possible for the consumer to process the data payload from the messages. As Theodolite uses Apache Kafka as message broker, as motivated by Henning and Hasselbring [2021b], we also use Kafka as data source and sink for our implementations.

2.7 The Containerization Technology Docker

Docker [Bernstein 2014] provides fast deployments of Linux applications inside portable containers. A container is a form of lightweight virtualization which are able to run in isolation regarding the underlying hardware. Furthermore, Docker isolates the containers view from the underlying operating system using namespaces. The image of a Docker
2. Foundations and Technologies

A container may contain only a basic set of operating system fundamentals or may contain a full application stack. Docker containers allow starting single instances of microservices at will. Docker allows a range of possible layering combinations. It may run directly on top of the underlying OS, within a virtual machine, or on top of a container orchestration framework. Thus, the deployment of Docker containers can be adapted to the given cloud environment.

2.8 The Cluster Manager Kubernetes

Kubernetes [Burns et al. 2016] is an open source cluster manager with support for Docker containers. Kubernetes decouples the application container from the details of the underlying resource management. A Kubernetes cluster consists of either physical or virtual machines called nodes. The containers get allocated to nodes as a management unit called pod. Pods can contain a group of containers and are addressable through unique IPs. This way, a pod can reach all other Pods in its network. The containers in a pod share their resources, such as data volumes. When a client requests a service, Kubernetes will forward the request to a pod. This allows Kubernetes to implement load balancing rules and distribute a request to a chosen Pod of a range of replicas.
Chapter 3

Theodolite Benchmark Implementation in Beam

In this section, we describe our implementation of the use cases, as detailed in Section 2.5.1. We will begin in Section 3.1 with describing how we read the input data and manage the watermarks using Beams Kafka connector. Following, we will gradually explain the implementation of the different use cases and our considerations in choosing the right one. We will focus on explaining the newly emerging concepts for each use case and omit to explain the same concepts multiple times.

3.1 Communication with Apache Kafka

Theodolite uses Kafka as a message broker between the workload generator and the application which is to be benchmarked. As such, our four use case implementations share the characteristic that each use case consumes data from Kafka topics. In order to configure a pipeline, such that it uses Kafka as a data source, Beam natively provides the class KafkaIO [Apache Software Foundation 2020b].

Read transformations, such as KafkaIO, can be applied to a pipeline and will return a PCollection representing the data in the consumed topic. Similarly, write transformations write the data of a PCollection to external data sinks such as files, file systems, databases, and messaging services.

Listing 3.1 shows our basic usage of KafkaIO. For better readability, we updated each consumer property with its own method call, in our implementation we combined the consumer properties beforehand as a map and only called updateConsumerProperties once. In UC3 and UC4, KafkaIO is used as shown.

That is, we apply the KafkaIO.read() transformation with the necessary configurations to our pipeline. Firstly, in Line 3 and 4, we specify the address of our Kafka instance and the topic that is to be consumed. Secondly, in Line 5–6, we need to set key and value deserializers. These classes are used to convert the byte array which is obtained from the topic into the correct data format. In our case, we read a key-value pair of the format String and ActivePowerRecord. To generate the deserializer for ActivePowerRecord, we use the Confluent Schema Registry [Confluent 2020]. This is a service which runs separate from Kafka and provides a RESTful interface for the storage and retrieval of, inter alia,
Avro schemas. The schema of a datatype defines the fields, of which the datatype is composed of. Our workload generator, upon sending the data to Kafka, notifies the schema registry of a new schema to be stored. When our Beam application instantiates the Kafka consumer for this topic, it will send a HTTP GET request to obtain the schema used for the configuration of the KafkaAvroDeserializer. This is done in Line 7–10, in updating the properties of our Kafka consumer. Lastly, we drop the Kafka metadata. These involve topic name, partition id, and offset since we do not need to keep this data for further processing. This changes the format of the forwarded PCollection from KafkaRecord of type String and ActivePowerRecord to a key-value pair of type String and ActivePowerRecord. In UC1 and UC2 we exchange withTimestampPolicyFactory with withProcessingTime() This sets the timestamp and watermark of the incoming record to the current system clock.

However, since we use event time, as introduced in Section 2.1.1, in UC3 and UC4, we need to add a custom TimeStampPolicy. In order to track the reading progress of a topic, Beam manages low watermarks for each partition. The low watermark of a streaming step is the timestamp of the oldest message not yet processed. This guarantees that every data before the low watermark has been processed. For every processing step, an input low watermark and an output low watermark is tracked. The input low watermark defines the oldest data that has not been sent to this step. The output low watermark defines the oldest data not yet processed in this step. This ensures correct aggregations as long as the watermark is increasing monotonously. There are different possible solutions for the approximation of watermarks. We will introduce two solutions we found to be suitable. As we can only use the available information, both solutions are a best-effort approach and will not yield the perfect approximation of time in every case.

```
final PTransform<PBegin, PCollection<KV<String, ActivePowerRecord>>> kafka =
  KafkaIO.<String, ActivePowerRecord>read()
    .withBootstrapServers("localhost:9092")
    .withTopic("input")
    .withKeyDeserializer(StringDeserializer.class)
    .withValueDeserializerAndCoder(KafkaAvroDeserializer.class, AvroCoder.of(
      ActivePowerRecord.class))
    .updateConsumerProperties(ImmutableMap.of("auto.offset.reset", "earliest"))
    .updateConsumerProperties(
      ImmutableMap.of("schema.registry.url", "http://localhost:8081"))
    .updateConsumerProperties(ImmutableMap.of("specific.avro.reader", "true"))
    .withTimestampPolicyFactory(
      (tp, previousWaterMark) -> new EventTimePolicy(previousWaterMark))
    .withoutMetadata();

pipeline.apply(kafka);
```
3.1. Communication with Apache Kafka

Listing 3.2. Naive Watermark Approximation

```java
public class NaiveWatermarkApproximation extends TimestampPolicy<String, ActivePowerRecord> {
    protected Instant currentWatermark;

    public NaiveWatermarkApproximation(final Optional<Instant> previousWatermark) {
        this.currentWatermark = previousWatermark.orElse(BoundedWindow.TIMESTAMP_MIN_VALUE);
    }

    @Override
    public Instant getTimestampForRecord(final PartitionContext ctx, final KafkaRecord<String, ActivePowerRecord> record) {
        this.currentWatermark = new Instant(record.getKV().getValue().getTimestamp());
        return this.currentWatermark;
    }

    @Override
    public Instant getWatermark(final PartitionContext ctx) {
        return this.currentWatermark;
    }
}
```

3.1.1 Naive Watermark Approximation

The apparent solution for the calculation of low watermarks for incoming data would be Listing 3.2. We define the lower bound of the watermark, which gets invoked if no knowledge about the time state of a partition is available, as the minimal value for the datatype `Instant`. The watermark is set to the lower bound upon resuming or first start of consuming data from a partition in Line 5. As new data arrives, we extract the timestamp of the `ActivePowerRecord` and set it as the timestamp for this key-value pair. After doing this, `getWatermark()` is invoked. This way we update the current watermark of the corresponding partition to reflect the timestamp of the last consumed `ActivePowerRecord`. We estimate the current time as the timestamp of the last consumed sensor reading. This implementation works correctly as long as all partitions deliver data. For partitions without data we can not estimate the watermark. This leads to a state in which the watermark is set as the lower bound until the first data arrives. Beam computes the output low watermark of the corresponding processing step for `KafkaIO` as the minimum of the watermarks of all partitions. Through downstream watermark propagation, our windowing functions will also have the lower bound as input low watermark and will not fire. This leads to data congestion at the window step, as no data will be forwarded.
3. Theodolite Benchmark Implementation in Beam

Listing 3.3. Idle Advancement

```java
public class IdleAdvancement extends TimestampPolicy<String, ActivePowerRecord> {
    protected Instant currentWatermark;
    private static final Duration IDLE_WATERMARK_DELTA = Duration.standardSeconds(2);

    public IdleAdvancement(final Optional<Instant> previousWatermark) {
        this.currentWatermark = previousWatermark.orElse(BoundedWindow.TIMESTAMP_MIN_VALUE);
    }

    @Override
    public Instant getTimestampForRecord(final PartitionContext ctx, final KafkaRecord<String, ActivePowerRecord> record) {
        this.currentWatermark = new Instant(record.getKV().getValue().getTimestamp());
        return this.currentWatermark;
    }

    @Override
    public Instant getWatermark(final PartitionContext ctx) {
        if (ctx.getMessageBacklog() == 0) {
            final Instant idleWatermark = ctx.getBacklogCheckTime().minus(IDLE_WATERMARK_DELTA);
            if (idleWatermark.isAfter(this.currentWatermark)) {
                this.currentWatermark = idleWatermark;
            }
        }
        return this.currentWatermark;
    }
}
```

3.1.2 Idle Advancement

In order to advance the watermark in case of missing data, we propose the method of Idle Advancement. The difference to the first solution is that we advance the watermark to current system clock time minus a delta in case of no new data arriving in a partition. In Line 19 we check if there is new data available in the partition. In the case of no new data arriving, we set the `idleWatermark` as the time at which we fetched the last message minus a constant delta. This delta is used to approximate the maximum delay of the creation of data and the ingestion into Kafka. To maintain the monotonously increasing nature of our watermark, we compare whether the `idleWatermark` is after the `currentWatermark`. In our
3.2. UC1: Database Storage

In the following, we explain our implementation of the use case Database Storage, as introduced in Section 2.5.1.

This use case topology is shown in Figure 3.1. Firstly, we apply the KafkaIO/read transformation, as explained in Section 3.1. Afterwards, we apply the MapElements transformation, in Line 1 of Listing 3.4, using an anonymous SimpleFunction. In Line 3–13, we define the function such that we map the value of our key-value pair as a GSON\(^1\) string encoding the corresponding information. One important thing to note is that we declare the gsonObj as transient. Using the modifier transient tells Java to ignore the original value of this object when serializing and instead set the default value. This is done so the SimpleFunction is still serializable. Since Beam forwards the function to the processing unit, it has to be serializable. Instantiating the gsonObj inside the apply method would introduce an overhead, since every incoming message would instantiate the gsonObj again. We chose not to use a real database, instead, we simply forward the results to the standard output. Theodolite’s benchmark UC1 does not use a real database, instead it forwards the result to the standard output. It is expected, that the database would be the bottleneck for this benchmark [Henning and Hasselbring 2021b]. However, it would be possible to use one of Beams many I/O connectors in order to write to databases, such as Cassandra and Elasticsearch [Apache Software Foundation 2020c].

---

\(^1\)https://github.com/google/gson
3. Theodolite Benchmark Implementation in Beam

Listing 3.4. Usage of KafkaIO

```java
/*apply(MapElements
     .via(
        new SimpleFunction<KV<String, ActivePowerRecord>, KV<String, String>>() {
          transient Gson gsonObj = new Gson();
          @Override
          public KV<String, String> apply(
            final KV<String, ActivePowerRecord> kv) {
            if (this.gsonObj == null) {
              this.gsonObj = new Gson();
            }
            final String gson = this.gsonObj.toJson(kv.getValue());
            return KV.of(kv.getKey(), gson);
          }})

```

Figure 3.2. Downsampling implementation

3.3 UC2: Downsampling

After applying the KafkaIO/read transformation, we use the Window.into() transformation. The window transformation must be supplied with a WindowFn. The WindowFn determines how records are grouped based on their timestamp and keys. We supply it with the provided FixedWindows function to create windows of a constant size. As we use processing time, we do not need to define additional triggering behaviour for the windows. Thus, the default trigger is used. The default trigger fires once when the watermark advances past the window bounds with no regard of late data. Looking at the aggregation step, we are confronted with two possible solutions. Firstly, we can use GroupByKey to obtain an iterable for every key. Then we can simply compute the aggregation results using a ParDo function. However, this solution brings a disadvantage with it. The aggregation for a given key is computed on a single machine in one thread. Large datasets can now lead to the machine running out of memory. Furthermore, in case of not evenly distributed keys, the keys which are much more frequent than others, so-called hot keys, might slow down the processing due to the worker with the hot key being busy while the workers with the regular keys are idle.

We circumvent this disadvantage by using Combine perKey. This gives us the possibility of combining a group of elements for a particular key in parallel. To apply this transformation,
3.4. UC3: Aggregation based on Time Attributes

we need to implement a subclass of CombineFn, as seen in Listing 3.5. The StatsAggregation
uses an accumulator to compute subsums of our final result. We have to implement four
methods contained inside the StatsAggregation. createAccumulator() is invoked for every
worker once per key in a window that is being distributed to it. To compute a subsum
upon getting new input, we use addInput, which simply adds the value to our sum. If
the computation for a key is distributed to multiple workers, mergeAccumulators is called
after every input element of the window has been processed. mergeAccumulators is used to
merge the subsums into the final result. Afterwards, we simply use extractOutput to obtain
and forward the single value representing the result. The advantage of this approach is the
possible higher parallelization of the aggregation step. However, this comes at the cost of a
network overhead, due to serialization and sending subsets of the data between worker
instances. Afterwards, we map the resulting key-value pair of type String and stats into a
key-value pair of type String and String. Lastly, the results are written into a Kafka topic
using the KafkaIO connector.

3.4 UC3: Aggregation based on Time Attributes

In the following, we explain our implementation of the use case Aggregation bases on
Time Attributes, as introduced in Section 2.5.1. The topology of this use case is shown in
Figure 3.3. Many of the applied transformations are already discussed in Section 3.2 and
Section 3.3. First we apply the KafkaIO transformation and read from the input topic as
described in Section 3.1. Different to the use case Database Storage and Downsampling
we use event time instead of processing time in this use case. Thus, we use a custom
timestamp policy, as described in Section 3.1 to extract the timestamp and set the water-
marks accordingly. The following map transformation is used to extract a time attribute
to generate a new key in form of HourOfDayKey, which combines the time attribute with
the sensor ID. The time attribute we set is the hour of the day. In the next step, we apply a
window transformation on the stream as seen in Listing 3.6. We use sliding windows with a
duration of 30 days and an advance period of one day. In this case we compute the average
over every hour of the day over a period of a month starting every day. To emit early
aggregation results we define a custom trigger. Beams default trigger fires exactly once,
when the watermark passes the end of the window. We still want to fire upon closing the
window, thus we configure our window with the trigger AfterWatermark.pastEndOfWindow().
withEarlyFirings allows us to combine this trigger with another trigger that might fire early
and repeatedly. We use a processing time trigger, which fires after the first element is added
3. Theodolite Benchmark Implementation in Beam

**Listing 3.5. CombineFn for ActivePowerRecord**

```java
public class StatsAggregation extends CombineFn<ActivePowerRecord, StatsAccumulator, Stats> implements Serializable{
    private static final long serialVersionUID = 1L;

    public StatsAccumulator createAccumulator() {
        return new StatsAccumulator();
    }

    public StatsAccumulator addInput(StatsAccumulator accum, ActivePowerRecord input) {
        accum.add(input.getValueInW());
        return accum;
    }

    public StatsAccumulator mergeAccumulators(Iterable<StatsAccumulator> accums) {
        StatsAccumulator merged = createAccumulator();
        for (StatsAccumulator accum : accums) {
            merged.addAll(accum.snapshot());
        }
        return merged;
    }

    public Stats extractOutput(StatsAccumulator accum) {
        return accum.snapshot();
    }
}
```

**Listing 3.6. Usage of the Window transformation**

```java
.apply(Window.<KV<HourOfDayKey, ActivePowerRecord>>into(SlidingWindows.of(duration).every(advance))
    .triggering(AfterWatermark.pastEndOfWindow())
    .withEarlyFirings(AfterProcessingTime.pastFirstElementInPane().plusDelayOf(triggerDelay))
    .withAllowedLateness(Duration.ZERO)
    .accumulatingFiredPanes());
```

3.5. UC4: Hierarchical Aggregation

Lastly, we discuss our implementation of the use case Hierarchical Aggregation, as introduced in Section 2.5.1. Henning and Hasselbring [2020a] presented a streaming topology for the Hierarchical Aggregation using the dual streaming model [Sax et al. 2018] and provided an implementation in Kafka Streams. We present our implementation using the Dataflow model. To implement this use case, we had to deviate from the original implementation in Kafka Streams as Beam’s Dataflow model uses a different approach for some of its most important transformations. Our pipeline is shown as a graph in Figure 3.4. For the sake of readability, we simplified the pipeline and merged more complex subgraphs into single nodes. Our implementation uses a bottom up approach to compute the results of every group in the hierarchy. For a hierarchy with depth \( n \), the first trigger firing would compute the results for all groups of level \( n \) and the following firing computes the results

to the window and a delay has passed. For our experiments we use a delay of 15 seconds. This trigger is reset after it fires and waits for the next element to be added to the window. We decided not to consider late data for our aggregations, as seen in Line 6. Lastly we configure the window to accumulate all early emitted results. The next transformation that we apply is the aggregation step. We use the same `Combine.PerKey` transformation as discussed in Section 3.3. Using the new generated `HourOfDayKey` we aggregate averages for every sensor ID and time attribute combination. Lastly, we map the results into the correct String format, containing only the identifier, and write it to Kafka using `KafkaIO.write`.

![Figure 3.4. Hierarchical Aggregation](image-url)
3. Theodolite Benchmark Implementation in Beam

for all groups of level \( n - 1 \). This is repeated until we arrive at level 1, at which point
we have computed the aggregations for all groups. In the following, we will explain our
implementation stepwise.

Generating the Hierarchy Changelog

Whenever an event which changes the sensor hierarchy occurs, we read the full newly gen-
erated hierarchy as a String from Kafka. Based on this hierarchy, we generate a changelog
which captures the relationship between each sensor and its parent groups. We begin
with applying a ParDo, with multiple outputs for a single element, in which we iterate
through the String representing the hierarchy and generate a pair consisting of each child
sensor and all its parent sensors as a set of Strings. We apply another ParDo, which uses a
state to compare the new child-parent pairs with already existing ones. In Beam, a state is
declared within the processing step for a single key in a window and thus, only state-cells
associated to a single key and window can be used. We use a state-cell containing the
parents for each child in order to detect changes in the hierarchy. In combination with
the Latest.perKey transformation, we only forward the newest parents for each child. As
the change in hierarchy could happen at arbitrary times, for example due to a manual
reconfiguration, we must use a global window with a custom trigger. This processing time
trigger will fire as soon as a new record is added to the global window and thus, it fires
after every reconfiguration. Furthermore, we defined the trigger to accumulate all fired
window panes. Thus, we capture every change in the hierarchy but only forward the latest
child-parent pair. Lastly, we apply a View.asMap transformation. This changes the datatype
from a window pane to an immutable PCollectionView to prepare it for being used as a side
input.

Flattening Sensor Readings and Aggregation Results

The sensor readings and results of earlier aggregations are read from Kafka using the
KafkaIO connector, as detailed in Section 3.1, using event time. Both streams are windowed
using a fixed window. The earlier calculated aggregations are furthermore converted from
AggregatedActivePowerRecord to ActivePowerRecord. In the reference implementation in Kafka
Streams and the original use case topology, as shown in Section 2.5.1, the window is applied
at a later step immediately before the aggregation. We apply the window earlier, as Beam
does not support joining streams with the default global windowing strategy or windows
with different sizing. To join both streams, we applied the flatten transformation on the
corresponding PCollections. Flatten merges multiple PCollections of the same type and
same non-global windowing strategy into a single PCollection. In the following, both sensor
readings and already computed aggregations are treated equally and we will refer to both
as sensor readings.
Building Pairs of every Sensor reading and Parent

In the next step, for every parent-child combination, we build a new record with a combined SensorParentKey, which consists of the identifier of the sensor and the corresponding parent, and the sensor reading of the child as value. In the reference topology, the streams containing the hierarchy were joined with the sensor data. However, even though Beam provides a join library\(^2\), it is not possible to join streams with different windows. As the use case requires us to use a global window for the hierarchies, we instead used Beam’s side input approach. A side input can be supplied to a ParDo to enrich the processing step with data from additional sources. Beam matches the window of the main input, in this case the sensor readings merged with the earlier results, with the corresponding window of the side input. As our side input uses global windowing, the latest fired window pane, which is used to build the view, will be matched with the window of the sensor readings. Our solution is adapted from the side input pattern Slowly updating global window side inputs.\(^3\)

The main difference is that we do not call for an update from inside the application. Instead, we update the side input whenever a reconfiguration is done outside our application. Inside the ParDo, we use the map provided by the side input to look up the corresponding entry for every sensor reading and forward a record for every pair of child and parent. In case of the hierarchy changing and the sensor being removed from a group, we forward a special record to indicate this change. Again, we use Latest.perKey to only forward the latest values as a basis for the aggregation step. The mayor drawback of using side inputs is however that the full map containing the sensor hierarchy is broadcast to all workers assigned this task. This might introduce a considerable overhead to our implementation.

Aggregation of Sensor Groups

At this point, we have generated records containing all sensors or sensor groups matched with their respective parent group as key and the last reading or aggregation result as value. Now, we aggregate the results for the next level of the hierarchy. Before we can apply the Combine.perKey transformation to calculate the aggregation results, we have to map the key to the parent group from the SensorParentKey. We have now generated a PCollection with records keyed by the sensor group they belong to and valued by the last sensor reading or aggregation result. Following this, we apply the Combine.perKey transformation supplied with a CombineFn as described in Section 3.3 and Section 3.4. This results in a record keyed by the sensor group and the result of the aggregation in form of a AggregatedActivePowerRecord. As we are not able to access the key of the current processed record in the CombineFn, we apply another transformation in which we set the identifier of the AggregatedActivePowerRecord equal to the corresponding sensor group. As we calculate the aggregation for the sensor hierarchy from the bottom up, we set a trigger to emit the result of the current level every 15 seconds.

\(^2\)https://beam.apache.org/documentation/sdks/java-extensions/  
\(^3\)https://beam.apache.org/documentation/patterns/side-inputs/
3. Theodolite Benchmark Implementation in Beam

When executing this Beam pipeline with the Samza runner, we discovered that applying an aggregation step on a PCollection that has multiple upstream inputs leads to a change in the time domain. This is due to Beam applying a CoGroupByKey transformation over multiple input collections. Due to this, the application changes from processing and event time to synchronized processing time, which is a time domain not yet supported by the Samza runner. To solve this problem, we reapplied the trigger after each step that uses CoGroupByKey.
Embedding the Use Cases into Theodolite

In this chapter, we explain the steps we made to execute our implementations, as described in Chapter 3, using the benchmarking framework Theodolite. We will begin in Section 4.1 with the adjustments to the execution routines of Theodolite. In Section 4.2 and Section 4.3, we will discuss the necessary steps to deploy a Flink and Samza cluster and run Beam pipelines on them.

4.1 Adjusting Theodolite to support the Execution of Beam Pipelines using the Flink and Samza Runners

Currently, the execution of experiments using Theodolite only supports a single stream processing engine as backend. Depending on the implementation chosen, this is either Kafka Streams or Apache Flink. The execution is shown in Figure 4.1. This diagram is simplified for the sake of readability.

The user starts the execution of an experiment using `theodolite.py` with the corresponding arguments to configure the use case, workloads, instances and further adjustments. In the example shown in Figure 4.1, UC1 is tested with a workload of 100 and a workload of 200 using a single instance. Inside the `theodolite.py` execution, the configurations are used to configure the experiment execution and call `run_uc.py` using the chosen subexperiment executor. This calls `run_uc.py` with the created configurations and further arguments for default configurations, which we omit at this point. In the following, we will simply label the configuration arguments as `arguments`. `run_uc.py` is responsible for the execution of a single subexperiment. In our case, we have two subexperiments. In the first subexperiment, we use a workload dimension of 100 and in the second subexperiment we use a workload dimension of 200.

Each subexperiment calls the same functions within `run_uc.py`. There are further setup functions that we will omit at this point for the sake of readability. These functions start the workload generator and the application for the benchmarks. After waiting for the benchmark duration, we run the evaluation on the results of said benchmark and reset the cluster for the following subexperiments. This procedure will be repeated for every subexperiment.
4. Embedding the Use Cases into Theodolite

Since starting and resetting the cluster requires framework specific adjustments, for example, to reset the cluster after running benchmarks using Samza as framework we need to delete Zookeeper nodes and Kafka topics used for the coordination of the instances, we chose to add a new parameter for `theodolite.py` which specifies the framework to be used. Based on this, we chose a subexperiment executor which runs the newly implemented program `run_uc_<spe>.py`. Where spe could be Flink or Samza. The drawback of this approach is that we will have to implement a new `run_uc_<spe>.py` for every added framework.
4.2. Deployment of Beam Pipelines in Kubernetes using the Samza Runner

However, this gives us the highest degree of flexibility regarding the deployment options and configurations of the underlying framework cluster. We added an implementation of `run_uc_beamflink.py` and `run_uc_beamsamza.py`. In both cases, we changed the implementation of `start_application` to set up the Flink or Samza cluster and start the execution of our application. We also adjusted the implementation of `reset_cluster` to tear down the according cluster gracefully. In Chapter 4, we will take a closer look at the necessary steps for the deployment of our application using the Flink and Samza runner.

### 4.2 Deployment of Beam Pipelines in Kubernetes using the Samza Runner

Depending on the underlying cluster resource management technology running the application, Samza offers different deployment options.

As Theodolite uses Kubernetes as cluster management system, we chose to run Samza in standalone mode instead of the Hadoop\(^1\) cluster based deployment option. In this mode, Samza uses Zookeeper and Kafka for the coordination of the instances, as seen in Figure 4.2. Running Samza in standalone, the user simply starts the amount of required instances. These instances will then use a leader election algorithm using Zookeeper and dynamically coordinate themselves. The following steps are executed when spawning new instances. First, the instance registers itself with Zookeeper as a new participant. If no leader is chosen, for example, in case of it being the first instance, it will become the new leader.

\(^1\)[https://hadoop.apache.org/](https://hadoop.apache.org/)
4. Embedding the Use Cases into Theodolite

leader. Every participant will now send a notification to the leader, which will in return recompute the assignment of tasks. The leader will then publish the new task assignment and the participants will use this assignment and begin processing. The leader will also reassign the tasks when a participant leaves the group, for example, because of failure or if the number of instances are decreased. The standalone mode allows us to use Kubernetes as our container orchestration system. Since Thedololite already uses Kubernetes, we chose this option to run Samza in standalone.

To deploy the application in standalone mode, we build a Docker image based on our application containing the Beam pipeline. We can use a Java 8 base image and add the distribution tar file together with the properties file for Samza. To use the Samza runner, we specify the Samza runner as a pipeline option and add the corresponding arguments to generate the properties from our properties file. We furthermore need to switch from local execution to standalone execution mode in the runner. For further configurations, we use environment variables which are set upon starting the instance. This Docker image acts as the basis for our Kubernetes deployment. We created a Kubernetes deployment resource with a container that runs the Docker images. By changing the Docker image running in the container, this Kubernetes deployment can be used for every use case. Scaling the replicas of this deployment will start multiple instances which coordinate themselves using Zookeeper.

4.3 Deployment of Beam Pipelines in Kubernetes using the Flink Runner

Flink’s architecture differs from Samza’s since it uses a dedicated central work coordinator node. The architecture of a Flink cluster is shown in Figure 4.3 A Flink cluster consists of three main components: the JobManager, the TaskManager, and the Flink Client. The JobManager is used as the central work coordinator of a Flink cluster. It monitors the TaskManager nodes and allocates tasks to each TaskManager. The TaskManager is the node, which actually performs the work that an application consists of. The Flink Client is responsible for compiling the application into a graph, which then is submitted to the JobManager using the provided REST API.

The JobManager can be run in three different modes: the application mode, the per-job mode, and the session mode. To the best of our knowledge, only the session mode is suitable for a Kubernetes deployment using a Beam application as a job.

Session mode allows one JobManager to accept multiple jobs and allocate the tasks of multiple jobs to the pool of TaskManagers. In this case, the job is submitted by a client outside the JobManager. If however one of multiple jobs running on the same JobManager brings down the JobManager due to an error, then all jobs running on the cluster will be affected. This downside does not affect us, since in our experiments we will only run a single job per cluster for a limited duration of time. One downside of the session mode
which does affect our experiments, however, is that we have to start and tear down a cluster after every subexperiment. This may introduce a considerable overhead and delay the initial processing of data. In our experiments, we take this overhead into account and use a delay of a minute until we consider the application to be running.

To deploy our application, Flink supports different deployment options. We chose to use a standalone Kubernetes deployment, since Theodolite uses Kubernetes and standalone allows us more direct control over the lifecycle and resource management of our Kubernetes deployments. In contrary to Samza, we use a total of five Kubernetes resources to deploy the Flink session mode cluster and submit our application.

To deploy the components of the cluster, as seen in Figure 4.3, we define a Kubernetes resource for each component. Furthermore, we define the number of task slots per TaskManager instance. As we aim to get comparable results with already executed experiments, we set the number of task slots per TaskManager to one. We use a single instance of the JobManager, as we do not need the improved fault tolerance of multiple JobManager instances. We furthermore set up a Kubernetes service for the JobManager and the TaskManager. The service is used to expose the ports for the communication between the JobManager and TaskManagers. It also allows us to access the web UI provided by the JobManager to monitor the cluster. To configure our cluster, we use a ConfigMap Kubernetes resource. The ConfigMap is mounted into the TaskManager and JobManager instances upon startup as a read only volume. Flink’s startup script allows to override the configurations and change the content of the volume provided by the ConfigMap. To support this behavior, we use an Init-Container which, before the application container starts, copies the configurations to a

Figure 4.3. Example Flink Cluster
4. Embedding the Use Cases into Theodolite

different read-write volume.

Lastly, we need to define the deployment of the Flink Client submitting the job to the JobManager. This is another Kubernetes resource of type deployment, which uses the REST API of the JobManager to submit the job. The container uses a Docker image that we build upon our Beam pipeline. Similar to the Samza Docker image, we use a Java 8 base image and add the distribution tar file containing our application. The submission is executed by running the application with the Flink runner providing the arguments containing the JobManager address and the degree of parallelism which defines the task slots per TaskManager. We furthermore enforce streaming mode instead of batch mode.
In this chapter, we will evaluate the results of our experiments on the scalability of Flink and Samza in combination with Beam. First we describe our experiment setup and explain the configurations for our experiments. Following, we will execute a range of preparatory experiments on different configurations, followed by our final experiments and discuss the results. A replication package and the results of our experiments are publicly available [Bensien 2021].

5.1 Experiment Setup

The experiments are executed in a private cloud with 4 nodes with each having $2 \times 16$ CPU cores with hyperthreading disabled and 384 GB RAM. The nodes are connected via 10 Gbits/s Ethernet. The cloud is run using Kubernetes. Further used software is shown in Table 5.1. For the Kafka client version used by the application, we manually chose the supported Kafka client version to prevent compatibility issues. We used the newest version of the Flink runner and the corresponding Flink version to run the cluster. For Samza, we chose 2.22.0 as the Samza runner version. The reason is that when upgrading to 2.27.0 we discovered a change in the Beam Kafka connector, introducing Beam operators, which are not yet supported by Samza.

5.2 Configurations for Experiments

As both Samza and Flink provide a wide range of configurations, which can be used to optimize the performance based on the use case, we decided to use the default configuration of both frameworks to gain comparable results. Due to not intended behavior of both runners, we made small adjustments to the default configurations. For the Flink runner we used the flag `--fasterCopy` to disable unneeded serialization and deserialization steps. We furthermore set the number of partitions equal to the number of instances. The reason for this and experiments supporting our decision are presented in Section 5.3. The last change we made was disabling Beam’s metrics in both runners, which is explained in Section 5.3.2.
5. Evaluation

Table 5.1. Software Stack

<table>
<thead>
<tr>
<th>Software</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kubernetes</td>
<td>1.18</td>
</tr>
<tr>
<td>Confluent Platform</td>
<td>5.4.0</td>
</tr>
<tr>
<td>Flink Runner</td>
<td>2.27.0</td>
</tr>
<tr>
<td>Flink Cluster</td>
<td>1.12.0</td>
</tr>
<tr>
<td>Samza Runner</td>
<td>2.22.0</td>
</tr>
<tr>
<td>Samza Cluster</td>
<td>1.3.0</td>
</tr>
<tr>
<td>Kafka Client for applications</td>
<td>0.11.0.2</td>
</tr>
<tr>
<td>Grafana</td>
<td>6.7.3</td>
</tr>
<tr>
<td>Prometheus</td>
<td>2.24.1</td>
</tr>
<tr>
<td>Kafka Lag Exporter</td>
<td>0.6.6</td>
</tr>
<tr>
<td>Java Runtime for Applications</td>
<td>8</td>
</tr>
</tbody>
</table>

5.3 Preparatory Experiments

We executed a range of preparatory experiments on different runner options for the Flink and Samza runner. We aimed at finding runner options to increase the scalability of our applications, while maintaining the default configurations of the underlying cluster.

5.3.1 Number of Partitions

As the partitions are being allocated to our application instances, the upper bound of the parallelism of our application is defined as \( \min(k, p) \), where \( k \) is the number of distinct keys and \( p \) is the number of Kafka partitions. To not restrict the parallelism, we have to use at least as many Kafka partitions as we deploy instances of the application.

On the one hand, Samza uses the container as the physical unit of parallelism which corresponds to the instances we start. On the other hand, Samza uses tasks as logical unit of parallelism. For each task in a container, a thread gets spawned to process said task. Tasks are managed implicitly by Samza itself. Samza creates a task for each partition of the input stream of an application. We were not able to configure this behavior of Samza. To test the impact of the amount of Kafka partitions on the performance of Samza, we executed experiments with the use case Database Storage. We chose Database Storage as it is the least complex use case and as such mechanisms such as state management, the communication between subtasks, and event time do not distort the results. We used a workload of 10 000 messages per second, which should be easy to handle even by a single instance. As configurations, we use Samzas default configuration with in-memory state. Later results confirmed that Samza was able to process this load with a single instance. As seen in Figure 5.1, we see a steady increase in the measured record lag. However, further experiments did show that when increasing the load, we were still able to process more
5.3. Preparatory Experiments

Figure 5.1. Queued messages using a workload of 10,000 data with a single instance. Thus, we draw the conclusion, that the increasing record lag is not caused by Samza not being able to process the amount of data, but has a different cause. When looking at the record lag of each particular partition, we saw that only some partitions were generating record lag. Even when running the experiment multiple times, we could monitor that the partitions with record lag remained the same. The cause of this behavior is not obvious to us. It might be caused by the scheduling of the threads, which represent the tasks, or how the polling of different partitions is implemented. When changing the amount of Kafka partitions to be equal to the amount of instances, we were able to reduce the slope to a negligible value. We saw that the load used in the first experiments could be handled by a single instance with a very slowly increasing record lag. In further tests, we changed the number of CPU cores per instance. We monitored that the number of partition has to be equal to the total number of CPU cores available for processing in order to reduce the slope of the lag record trend.

As we aim to gain comparable results to already executed experiments by Henning and Hasselbring [2021b], we execute our experiments with a single thread per instance. To achieve this, we set the number of Kafka partitions equal to the number of instances for all following experiments.

5.3.2 The Impact of Beam’s Metrics on Scalability

We decided to benchmark the impact of enabling Beam’s metrics using the Samza runner. Beam’s metrics are additional metrics, provided by the runner, beyond the metrics provided by the SPE. By default, Beam metrics are enabled. On the one hand, benchmarking the impact of disabling Beam metrics can give us insight about the overhead produced by using Beam. On the other hand, Zhang et al. [2020] highlighted the negative impact of a bug causing the Samza runner to emit the Beam metrics four times as often as intended.
5. Evaluation

Unfortunately, this was only fixed in the newest version of the Samza runner (2.27.0). A bug which was introduced after version 2.25.0 regarding the Kafka connector prevents us from using this version. We executed this Benchmark using Samza’s default configuration with one partition per instance and executing the use case Database Storage. We furthermore used in-memory state.

As seen in Figure 5.2, a high overhead is introduced using the Beam metrics in version 2.22.0 of the Samza runner. Without Beam metrics enabled, we could process a workload of 25 000 messages per second using a single instance. When we activated Beam metrics, the needed instances to process a workload of 25 000 increased to three. We can still conclude that in both cases, the processed workload scales linear in relation to the instances. We benchmarked up to a number of 20 instances. With Beam metrics enabled, we were not able to process loads higher than 200 000 messages per second. As the metrics provided by Samza itself will still be accessible, we see it reasonable to deactivate the Beam metrics. We conclude that enabling the Beam metrics introduces a high overhead but does not prevent us from scaling the number of instances accordingly to compensate this overhead. In the following experiments we will disable Beam’s metrics.

5.3.3 The Impact of fasterCopy and Beam’s Metrics on the Scalability of the Flink Runner

When using native Flink, both a serialization step and a deserialization step is performed to copy the data between every operator applied on the input stream. However, Beam’s model includes that each PCollection is immutable. Because of the immutability of the PCollection, the copy operation between operators in Flink is not needed when using Beam’s model and the following operator can simply reuse the already existing PCollection instance, relying
5.3. Preparatory Experiments

![Graph showing demand for UC1 using the Flink Runner with multiple Configurations](image)

**Figure 5.3.** Demand for UC1 using the Flink Runner with multiple Configurations

on the assumption that the former operator does not alter the `PCollection`. This relies on the user fully adapting the Beam model. Using the *Direct runner*, we tested that our use cases do indeed comply with the Beam model and do not violate the immutability of the `PCollections`. The *Direct runner* performs a range of checks to ensure that the pipeline only relies on semantics that are part of the Beam model.\(^1\) Thus, we benchmarked the impact of removing the unneeded copy step between operators using the `--fasterCopy` flag. We also benchmarked the impact of disabling Beam metrics in the Flink runner. The results are shown in Figure 5.2. While we are aware that a bug increased the overhead of enabling Beam metrics in the Samza runner, as depicted in Figure 5.2, we are not aware of such a bug in the Flink runner. We can see that disabling the Beam metrics does not increase the performance of the Flink runner. On the contrary, for some workloads, the performance is better with Beam metrics enabled. As we see no reason for this to be the case, we expect this to be due to fluctuations in different experiment executions. In Section 5.4.1 we will explain our observation regarding the Kafka partition assignment of the Flink runner, which might cause different results for multiple executions of the same experiment. We can however see a high performance increase when enabling `--fasterCopy`. While a single instance without `--fasterCopy` enabled could only process a workload of 50 000, when enabling `--fasterCopy` we increased the workload that a single instance could handle to 125 000. For higher loads, the performance was still substantially better, for example, a workload of 400 000 could be handled with approximately half as many instances as before. Thus, we decided to use the flag `--fasterCopy` for our experiments. Since a bug forced us to deactivate the Beam metrics for the Samza runner, we also deactivated them for the Flink runner to gain more comparable results.

\(^1\)https://beam.apache.org/documentation/runners/direct/
5. Evaluation

5.4 Results and Discussion

In the following, we will present the results of our scalability benchmarks executing the four use cases introduced in Section 2.5.1 and executed with the Flink and Samza runner for Beam.

5.4.1 UC1: Database Storage

In Figure 5.4, we see the scalability of the Flink runner in comparison to the Samza runner when executing the use case Database Storage. We benchmarked workloads ranging from 25 000 up to 800 000 messages per second and provided up to 50 processing instances. We can see that a single instance using the Flink runner is able to process a workload of up to 150 000, while the Samza runner is only able to process 25 000 using a single instance. The Samza runner scales according to our observations regarding the performance of a single instance. We see that while 1 instance can process a workload of 25 000, 4 instances can process a workload of 100 000. Using higher workloads, we observe that the workload-instance ratio is even increasing. For a workload of 400 000 we only need 12 instances and for a workload of 800 000 we need 20 instances instead of the expected 16 and 32. We can see that the overhead of running multiple instances does not reduce the performance for the Samza runner in this use case. This was expected, as our operations can be executed completely asynchronous without communication between workers. If we however look at the Flink runner, we can observe that while it performs better for low loads, it does not scale accordingly. At lower workloads, the Flink runner outperforms the Samza runner by a large margin. While we need 5 instances for a workload of 150 000 with the Samza runner, a single instance of the Flink runner is able to process the same load. While the
workload-instance ratio is at 150,000 per instance early on, it drops continuously to as low as 40,000 per instance at a workload of 400,000. When looking deeper at the cause of this, we observed that Flink distributes the Kafka partitions unevenly across the Taskmanagers. While every taskslot is used early on, after a fixed duration, some taskslots returned an idle state. We could however see, that every input partition was being read from. This leads us to the conclusion that either the Beam Kafka connector or Flink itself does not allocate an input partition to every Taskmanager. The documentation of KafkaIO [Apache Software Foundation 2020c] states that partitions get assigned evenly across workers. If this is the case, a possible reason could be Flink calling a shuffle or rebalance operation on the Kafka partitions. These might lead to the expected behavior. We were not able to test this, since we do not use Flink’s native API, where these operations are accessible. Beginning at a workload of 450,000, we can see that both runners perform close to equal. At workloads of 500,000, 625,000, 650,000, and 750,000 the Samza runner actually outperforms the Flink runner. If the workload-instance ratio continues to drop for the Flink runner, we suspect the Samza runner to outperform the Flink runner at higher workloads. However, as the instance count using the Flink runner does not directly translate to the non-idling Taskmanager count, we have to keep in mind that this conclusion might change when the input partitions are assigned evenly.

5.4.2 UC2: Downsampling

In Figure 5.5 we can observe the results of our scalability benchmark of the use case Downsampling. We benchmarked workloads ranging from 5,000 up to 75,000 using steps of 5,000. In this use case, the Flink runner outperforms or performs the same as Samza runner for every workload. When using the Flink runner, we observe that, while the partitions
5. Evaluation

still are distributed unevenly, no taskmanagers idle completely due to more subtasks being available. The Flink runner might also perform better when using lower instances. When benchmarking UC1, we used up to 20 instances using the Flink runner, while in this benchmark, only 7 instances were needed. We can however observe that the workload to instance ratio still worsened when more instances were used. With a single instance, the Flink runner could process a workload of 20 000, while the ratio decreased to around 10 000 per used instance when benchmarking a workload of 75 000. This time, we could also see this decrease, even though weaker, when using the Samza runner. A single instance could process the workload of 10 000, but to process a workload of 75 000, we needed 9 instances.

5.4.3 UC3: Aggregation based on Time Attributes

In Figure 5.6, the results of our experiments for the use case Aggregation based on Time Attributes are shown. For this use case, we benchmarked up to 50 instances and workloads ranging from 250 to 5 000 using steps of 250. We also adjusted the service level objective for when to mark a subexperiment as successful to a maximum record lag slope of 250. For both the Flink and the Samza runner, we were not able to find a difference in performance when using in-memory state in comparison to using RocksDB as state backend. We expect that the managed state was small enough to fit in memory for all subexperiments. As RocksDB still uses an in-memory caching layer, the results are not unexpected as long as the application does not need to fall back to on-disk storage. The only outlier of this is the Flink runner at a workload of 1 250, which could be processed by 18 instances using in-memory state but needed 30 instances using RocksDB. We can however observe that the Samza runner performs far better in comparison to the Flink runner. While we could
5.4. Results and Discussion

![Graph showing number of instances vs nested groups for Flink and Samza runners.]

Figure 5.7. Demand for UC4

still process a workload of 5,000 using 14 instances with the Samza runner, we could only process a workload of 750 with the same number of instances and were not able to process a workload of 2,000 using up to 50 instances with the Flink runner.

5.4.4 UC4: Hierarchical Aggregation

We executed experiments with hierarchies with up to 7 nested groups for the use case Hierarchical Aggregation. For our experiments with 5 nested groups, we applied a maximum record lag slope of 100, as only around 1000 sensor readings are generated per second. We used the default of a maximum slope of 2000 for the experiments with 6 or 7 nested groups. As we expect the Hierarchical Aggregation to use state to a greater extent, we enabled RocksDB as state backend for both runners. Our experiments on UC3 did show us that even when the state is small enough to fit into memory, we could not measure an overhead when using RocksDB. We provided up to 100 instances for our benchmarks. We used a window size of 60 seconds to group the sensor readings and forwarded results every 30 seconds. Thus, we are still able to calculate the full hierarchy for the first window. We can see that both frameworks are able to process a hierarchy with 5 nested groups using 5 instances. However, the Flink runner outperforms the Samza runner when using hierarchies with 6 or 7 nested groups. At 6 nested groups the Samza runner already needed 35 instances to process the workload, while the Flink runner only needed 10. We were not able to process hierarchies with 7 nested groups with up to 100 instances of the Samza runner, while we were still able to process this workload with 65 instances of the Flink runner.
5. Evaluation

5.5 Threats to Validity

When observing the state of each subtask in the Flink runner experiments, we found the unexpected behaviour of an uneven partition assignment, as described in Section 5.4.1. We were not able to find the culprit of this behavior and were able to observe it in all our experiments with the Flink runner. We can not rule out that this behavior is introduced by our setup. Experiments where all partitions are assigned equally across the instances might produce results with a better performance of the Flink runner.

As both Flink and Samza offer a large range of configuration options, we only benchmarked the default cluster settings. Using other configurations might increase the performance of both stream processing backends. However, finding the most performant configuration is not within the scope of this thesis, as it would need a lot of time, as most of our experiments ran for multiple hours. Thus, our findings are limited to the comparison of the two frameworks without further optimization.

To improve certainty of our findings, we would also need to run the experiments multiple times to exclude possible deviations between different experiment executions. Additionally, other cloud environments could be used for further experiments to rule out that the hardware influenced our results.

As Samza uses Kafka for the communication between tasks, further experiments using different Kafka configurations might lead to better results. We could for example increase the resources available to Kafka, as it might be under heavier load when running experiments with the Samza runner.

When using Apache Beam as abstraction layer to implement a use case once and execute experiments with different stream processing backends, certain aspects have to be considered. Using the same implementation for different backends eliminates possible performance differences introduced by deviating APIs. However, the results are limited in scope, as the Beam runner for a stream processing engine may introduce a considerable overhead. The overhead for different use cases is not predictable and the findings may not translate to the comparison of native implementations [Hesse et al. 2019].
Chapter 6

Related Work

As the development of stream processing technologies gained increasingly more attention in the last years, there has been research regarding the performance of different stream processing approaches. The research focussed on measuring the throughput and latency of stream processing applications. The approach we took to measure the horizontal scalability of stream processing engines, has only been executed before by Henning and Hasselbring [2021b], using Flink and Kafka Streams as benchmarked SPEs. This work laid the basis for our thesis and we adopted the used approach.

Tucker et al. [2008] introduced a benchmark for queries over data streams, based on a scenario of online auctions. Similar to our approach, this is based on real-life use cases. Their use cases include filter, aggregation, and join queries. Measurements have been done on how fast the system can process the data and how accurate the data is. This is measured using a metric called output matching. Output matching compares the difference between the ideal system output and the benchmarked implementation. This was adopted as the NexMark benchmark suite\(^1\) for benchmarking Beam runners.

Li et al. [2018] described challenges and experiences occurring while developing the IBM Streams runner\(^2\) for Beam. The authors discuss how handling windows and state can efficiently be realized using IBM Streams as the backend. To evaluate the optimizations in the pipeline translation, benchmarks using the NexMark benchmark suite have been conducted. Additionally, micro-benchmarks which focus on the performance of the ParDo and GroupByKey operations were performed. The benchmarks focussed on measuring the latency and throughput of Beam applications using the IBM Streams runner. The results were used to compare the IBM Streams runner, the Flink runner, and the Spark runner.

Work on benchmarking the Samza runner has been done by Zhang et al. [2020]. Zhang et al. [2020] benchmarked the performance of the Samza runner in comparison to the Flink runner, the Direct runner, and native Samza using the Nexmark benchmark suite, similarly to Li et al. [2018]. It was possible to identify bottlenecks by inspecting the CPU usage of different processing stages. The usage of Beam’s metrics was found to have a huge impact on performance. Disabling Beam’s metrics increased the throughput by a factor of 12. On the one hand, this was due to a bug which caused the metrics to be called four times for every bundle instead of once. On the other hand, the metrics updated too often and the update itself was unnecessarily CPU intense. These bugs have been fixed with version

---

\(^1\)https://beam.apache.org/documentation/sdks/java/testing/nexmark/

\(^2\)https://ibmstreams.github.io/streamsx.documentation/docs/beamrunner/
6. Related Work

2.27.0 of the Samza runner.

Stamatakis and Samoladas [2019] proposes a high-level real-time query optimization platform which, based on metrics collected from different SPEs, assigns queries to the most suitable SPE. The used SPEs are Kafka Streams and Beam in combination with Flink, Spark, and Apache Apex. Benchmarks on the throughput and latency of the runners of Flink, Spark, and Apache Apex are executed. The results show that out of these 3 SPEs, Spark performs worst on streaming data, while the benchmarks on Spark and Apache Apex obtain comparable results.

Halas [2017] researched the applicability of stream processing technologies on network flow processing monitoring and analysis. The focus was set on measuring the stream processing performance of Apache Spark on network traffic data using performance benchmarks in a distributed environment based on real datasets. Similar to our results when benchmarking the Samza runner, [Halas 2017] found that increasing the input partitions beyond the number of instances decreased the performance.

An evaluation of Apache Beam as abstraction layer for stream processing systems has been presented by Hesse et al. [2019]. Hesse et al. [2019] compared the Beam runners for Apache Flink, Apache Spark, and Apache Apex with their native counterparts. To keep the measurement application and system independent, the performance is measured using a result calculator which queries Kafka, instead of using the metrics provided by the stream processing systems. While this is similar to our measurement approach, the metric used is the execution time based on Kafka's timestamps, instead of the record lag we used. As use cases, four different stateless queries were used. Hesse et al. [2019] came to the conclusion, that the abstraction layer Apache Beam introduced a noticeable impact on the performance. In comparison to native applications, the applications implemented using Beam were slower with a factor of up to 58. This led the authors to two conclusions. Firstly, using Beam comes at the cost of worse runtime performance compared to native implementations. Secondly, using Beam to benchmark the underlying stream processing framework will lead to different results than using native implementations. Most importantly, the performance penalty varies among use cases and is currently not predictable.

These works measured the performance of stream processing frameworks in particular through analyzing the throughput and latency of a system. We applied a different measuring method, as presented in Section 2.5.2. Thus, we did not investigate the performance of single processing stages of our application, as for example Zhang et al. [2020] did on his work on the Samza runner, but rather measured the scalability of a fully deployed microservice application.

Similar to our work, Biernat [2020] applied Theodolite’s benchmarking method to evaluate the scalability of Apache Flink. Biernat [2020] provided implementations of the same use cases using Flink’s native API and executed comparable experiments using the demand metric. In contrast to our approach, the focus was set on identifying and evaluating different configurations of Flink, namely the checkpointing interval, the number

---

3 https://apex.apache.org/
of taskslots per instance and different state backends. The conducted experiments led the author to the conclusion that the checkpointing interval does not have a large impact on the scalability of Flink. Providing a higher number of taskslots led to the expected increase in performance except for UC2. Furthermore, using RocksDB as state backend proved to increase the performance in UC3 and 4 but decrease the performance in UC3. This is different from our findings, where the results for in-memory and RocksDB state backends were similar in UC3.
Chapter 7

Conclusions and Future Work

In this chapter, we will draw the final conclusions on our work and evaluate whether we achieved our goals. Additionally, we will provide an outlook on future work on the topic of benchmarking the stream processing backends of Apache Beam.

7.1 Conclusions

The aim of this thesis was to apply the Theodolite benchmarking method to evaluate the scalability of Apache Beam using Apache Flink and Apache Samza as processing backends. We set three goals in Section 1.2.

The first goal we achieved was to lay the foundation for our benchmarks, by implementing four common use cases using Beam’s programming model. For UC1 and UC2, it was not necessary to change the original use case topology. For UC3 and UC4, we were forced to deviate from the reference implementation using the dual streaming model to secure comparability in the results of the benchmarking experiments. In UC3, we were able to implement the original topology, however, Beam’s windowing and triggering model differed from the reference implementation. While Kafka Streams continually publishes interim results which get refined over time, we had to manually configure a trigger to emulate this behavior. More severe changes were made to the topology of UC4. Again, we were confronted with the more strict approach to windowing in Beam, compared to the original topology. This made it necessary to apply the aggregation window and trigger at an earlier stage. Another consequence of Beam’s windowing model is that instead of applying a join transformation, we used Beam’s side input mechanism to enrich the input stream with information about the hierarchy. In addition to laying the groundwork for our experiments, we add to the original implementation using the dual streaming model and provide an implementation using Beam’s Dataflow Model, with our implementation of the Hierarchical Aggregation.

The second achievement of this work is the embedding of the use cases into the framework Theodolite and the provision of the necessary infrastructure to execute experiments based on our implementation. In order to achieve this, we added the possibility for the user to decide between the different supported stream processing frameworks when executing experiments. For this, we implemented the setup and teardown routine for a Samza and Flink cluster in a cloud environment. For Flink, this included multiple Kubernetes
resources to set up a cluster which we submit our application job to. Samza only relied on a single Kubernetes resource, but uses the Kafka cluster for the coordination of the instances. Thus, the tear down of a Samza cluster included that we reset Kafka and Zookeeper to a clean state. Using these adaptations to Theodolite, it is possible to add new use cases implemented in Beam and deploy them on a Kubernetes cloud using the Samza and Flink runner. Furthermore, it is possible to use different runners by adding the cluster setup and teardown routine as new python programs.

The third goal was to execute experiments on the horizontal scalability of the Flink and Samza runner for Beam. Our experiments resulted in the finding that the Flink runner performs better than the Samza runner for UC1, UC2, and UC4. In UC3, the Flink runner unexpectedly performed worse than the Samza runner by a large margin. The reason for this might be the overhead introduced when using Beam as another abstraction layer. Related to this, Hesse et al. [2019] found that using the Flink runner for Beam in contrast to native Beam applications introduced unpredictable overhead. The author’s findings where heavily dependent on the use cases that were benchmarked. Based on this, our UC3 might be a use case where the Flink runner introduces a high overhead while the other use cases introduce a smaller overhead. Experiments like ours can help with finding these overheads and comparing different runners to find the most suitable for a given use case.

7.2 Future Work

In our experiments, we focussed on comparing the Beam runners for Apache Flink and Apache Samza using default configurations. In real world deployments, the performance can be improved by changing the configurations. This includes the memory available for each process and the number of threads available for each instance. Increasing the parallelism and resources per instance could give valuable insight at which workloads and number of instances vertical or horizontal scaling is more efficient. Furthermore, the default configurations of Flink and Samza can themselves differ from each other. For example, in Beam PCollections are processed in subsets called bundles, and when using default configurations, the maximum bundle size for the Flink runner is set to 1000, while the Samza runner sets the bundle size to 1. When running preparatory experiments, we found that even smaller configuration changes, which only disabled unwanted behavior of the runner, already had a big impact on the performance of both runners. Optimizing the configurations to gain the highest performance would include executing experiments to compare different configurations.

By default, Beam disables the checkpointing mechanism of Flink and Samza. Thus, our applications were tested without fault-tolerance. On the one hand, this does not translate to real world settings where fault tolerance is of utmost importance. On the other hand, our results are only comparable to a limited extend with earlier conducted experiments on native Flink and Kafka Streams. Thus, we propose that further experiments with checkpointing enabled would lead to more applicable results. One of the main differences
of Flink and Samza is that Flink offers exactly-once processing guarantees, while Samza offers at-least-once processing guarantees. This is the result of different fault recovery mechanisms, as described in Section 2.4.3. On the one hand, this would require us to induce faults into the system. On the other hand, we would need to run the subexperiments over extended periods of time, due to the size of the state to be restored influencing the recovery time. This could however give us valuable insight into the potential advantages and disadvantages of both approaches.

In the future, the provided implementations can be used to execute benchmarks with other Beam runners. This might include running benchmarks on the Hazelcast Jet [Gencer et al. 2021] runner and the Apache Spark [Zaharia et al. 2016] runner. However, currently the Hazelcast Jet runner is still in an experimental state and does not make use of all capabilities present in Hazelcast Jet. Similarly, the Apache Spark runner is also in an experimental state and when using Spark’s Structured Streaming it only supports batch processing. As soon as these runners exit the experimental state, they offer interesting benchmarking opportunities.

Due to issues with the newest version of the Samza runner, we were limited to the older version 2.22.0. As the newer versions of the Samza runner include performance optimizations, a repetition of our experiments could lead to better results for the Samza runner.

Insight into the performance of different stream processing engines in other application domains could be gained through addition of different use cases. This might include the newly emerging field of using stream processing approaches to apply machine learning algorithms on streams. Possible use cases might be found in the field of preprocessing data before applying machine learning algorithms or using stream processing frameworks to run the machine learning algorithm itself [Gomes et al. 2019]. For example, Apache Spark provides an own machine learning library.
Bibliography


Bibliography


Bibliography


