Scalability Benchmarking of Apache Flink

Bachelor’s Thesis

Nico Alexander Biernat

September 29, 2020

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 selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 29. September 2020
Abstract

In a time where more and more data is generated, collected and processed, the ability to process Big Data in real-time has become more relevant for faster reaction to important events in several application domains. This led to the development of numerous distributed stream processing engines that support the development of large streaming applications. An important characteristic of a stream processing engine is its ability to scale in order to handle an increasing workload.

In this thesis, we extend the first work on benchmarking the scalability of distributed stream processing engines by providing the necessary components to execute scalability benchmarks for the stream processing engine Apache Flink. For this, we migrate common use cases for stream processing, deriving from an Industrial Internet of Things application, to Apache Flink. Furthermore, we integrate those implementations into an existing framework for scalability benchmarking. The scalability benchmarks are executed in a public cloud environment and the effect of different configurations on the scalability of Apache Flink are evaluated. Our results show that while some configurations positively impact the scalability of Apache Flink for one use case, they might be less beneficial for or even negatively impact the scalability for another use case.
1.1 Motivation

Stream processing has gained a lot of popularity in the fields of Big Data and the Internet of Things and certainly has many benefits over batch processing for specific use cases.

But there are so many different stream processing frameworks available that choosing the best suiting one for a project might turn out to be quite difficult. Also most stream processing frameworks offer a large amount of configuration options which can be specified to optimize for goals such as performance or reliability.

Some criteria for choosing a stream processing engine are the compatibility to other employed technologies such as messaging systems or database systems, the support of specific programming languages or programming paradigms, and the availability of certain features. Moreover, performance aspects such as throughput and latency are important for very large or time critical applications as well as the ability to scale properly with increasing load on the system. Therefore, we need benchmarks to compare the performance of stream processing engines.

A lot of work has already been published on performance benchmarking of different stream processing engines considering the performance metrics of throughput and latency [Karimov et al. 2018; Chintapalli et al. 2016; Shahverdi et al. 2019; Hanif et al. 2019; Lu et al. 2014]. But none of that work focused especially on the scalability of stream processing engines so far. This thesis is based on and extends the first paper on scalability benchmarking of stream processing engines [Henning and Hasselbring 2020].
1. Introduction

1.2 Goals

1.2.1 G1: Implementation of Use Cases for Stream Processing in Apache Flink

Our first goal is to implement four common use cases for stream processing using Apache Flink. The use cases are adopted from Henning and Hasselbring [2020] and derive from an analytics platform for monitoring the energy consumption of industrial environments [Henning et al. 2019]. The use cases are database storage, hierarchical sensor aggregation, downsampling and aggregation based on time attributes. In Chapter 3 we are going to describe the use cases and their implementation in detail.

1.2.2 G2: Instrumenting the Use Case Implementations as Scalability Benchmarks using the Theodolite Benchmark Framework

In their paper, Henning and Hasselbring [2020] propose a framework for executing scalability benchmarks of stream processing engines. Our next goal is to embed the implemented use cases from G1 into the Theodolite framework such that they can be instrumented as scalability benchmarks for Apache Flink. Consequently, we need to choose a fitting deployment option for Flink and investigate how to measure scalability as described by Henning and Hasselbring [2020] for the concrete case of Flink.

1.2.3 G3: Execution and Evaluation of Scalability Benchmarks for Different Configurations of Apache Flink

Multiple benchmarks based on G2 will be executed for different workloads and numbers of instances with different configurations of Apache Flink to find out how a particular configuration will affect the scalability for a specific workload dimension. Execution will take place in a comparable cloud environment such that benchmarking results from other stream processing engines or configurations can be compared in order to determine which stream processing engine or configuration scales up best with a specific use case. Finally, the results of the scalability benchmarks will be evaluated, visualized, and discussed.

1.3 Document Structure

The required foundations and used technologies will be explained in Chapter 2. We then continue with our approach in Chapter 3 which contains the migration of usecases to Flink, the deployment of the usecases and their execution. Further, we are going to identify suitable metrics for benchmarking and configurations of Flink that we want to evaluate. In Chapter 4 we are going to describe our evaluation method and setup as well as evaluate
1.3. Document Structure

and discuss the results of the benchmarks. We then list related work in Chapter 5 and draw conclusions and give an overview of future work in Chapter 6.
In this chapter we will explain the foundations and present the technologies that are required for the entire thesis. We lay the focus of the foundations on scalability, stream processing and the benchmarking method that we are going to use. Apache Flink will be the technology that we will explain in most detail, as it is part of the main subject of this thesis.

2.1 Scalability of Software Systems

Scalability is defined as the ability of a software system to adapt to increasing load on it such that it can keep functioning as intended [Smith and Williams 2001]. Two categories of scaling are distinguished.

**Vertical Scaling**

With vertical scaling the computer that is running the software can be upgraded to or replaced by a more performant one. The software can then use the added system resources to handle more load. This can only be done to a certain degree as a single computer’s upgradeability is limited.

**Horizontal Scaling**

Horizontal scaling addresses this issue by adding more computers that cooperate through the use of a network to handle the increasing load on the system. This method of scaling is not limited directly like vertical scaling but it requires a lot more work on the software side.

Some inherent problems of distributed systems that have to be overcome are consistency of data, consensus on transactions, synchronization between computers, distributing the workload evenly, and failure of computers or the network.

But the benefit of possibly unbounded scalability that horizontal scaling offers has led to the development of complex software systems and frameworks some of which we are going to use or benchmark in this thesis. Horizontal scaling can be achieved by three paradigms which are described in the scale cube [Abbott and Fisher 2009].
2. Foundations and Technologies

First, there is horizontal duplication where multiple instances of a program are running in parallel usually behind a load balancer.

Second is functional decomposition where the functionality of a software system is split up in such a way that the smaller parts can be run individually. Service oriented architectures and especially microservice architectures are typical applications of this approach.

And third, we have data partitioning in which multiple instances of a program are running like in horizontal duplication but each instance is only responsible for a subset of the data that is being processed. Data partitioning is an important concept of stream processing engines and distributed message brokers such as Apache Flink and Apache Kafka.

In this thesis, mostly horizontal scaling is regarded and for simplicity we will use the term scalability interchangeably for horizontal scalability. Although, we are going to evaluate to some degree how additional vertical scaling affects the horizontal scalability.

2.2 Stream Processing

Stream processing is a programming paradigm for processing unbounded streams of data online or in other words (soft) real time. Contrary to offline batch processing there is no need for storing the whole dataset before processing it.

A data stream is an infinite series of records commonly consisting of a key, value, and timestamp on which stream processing applications can execute queries and transformations. Usually a messaging system is involved to manage the data streams as topics and split those topics up into partitions based on the records primary key. The partitioning of the data allows the stream processing application, which is usually built upon a framework such as Apache Flink, to parallelize the processing of that data. A stream processing application then executes a series of user defined operations, that form a directed acyclic graph from data sources to sinks.

Benefits of stream processing over batch processing include the ability to analyze and react to events faster, the missing necessity for storing incoming data before processing it, and the fact that data often already comes in the form of infinite streams for example with sensor readings in the Internet of Things.

2.3 Big Data and the Internet of Things

Big Data is the term used for data with such a high volume, velocity or variety that special technologies and methods are required for the processing of that data [De Mauro et al. 2016].

The Internet of Things (IoT) is the concept of connecting more devices than just computers to the internet. In many areas the Internet of Things has already been established.
Households are equipped with smart home devices like lights, thermostats and smart meters that can be controlled and monitored via the internet. Industrial production environments also use smart meters and other IoT devices to monitor their machines in real time (Industrial Internet of Things IIoT). This allows for faster reaction in case of a machine failure or another problem.

The use cases that the benchmarks in this thesis are based upon are prevalent in the processing of big data generated by smart meters in industrial production environments.

### 2.4 Benchmarks

A benchmark is a standard tool for the evaluation and comparison of competing systems. The key characteristics of a benchmark are its relevance to actual applications of the benchmarked system, the reproducibility of benchmarking results amongst multiple runs with the same test configuration, its fairness or the ability of different configurations to compete on their benchmark results, the verifiability or accuracy of the benchmarks results, and its usability for users to run the benchmark in their test environment [v. Kistowski et al. 2015].

To achieve these characteristics for a benchmark it is necessary to design use cases and workloads that are as similar as possible to real applications and workloads as well as employ specific performance metrics that ensure the reproducibility and fairness of the benchmark results. Despite the best effort of achieving the above mentioned characteristics, benchmarks still are a form of experiment and therefore susceptible to unexpected influences which can be threats to the validity of the results.

### 2.5 The Theodolite Scalability Benchmarking Method

Theodolite is the first and currently only method for benchmarking the scalability of stream processing engines. In their paper, Henning and Hasselbring [2020] identify 4 use cases for stream processing that are derived from an Industrial Internet of Things analytics platform for industrial power consumption monitoring [Henning et al. 2019]. These four use cases are database storage, hierarchical aggregation of sensor data, downsampling of a record stream and an aggregation of sensor data based on time attributes. A detailed explanation of these use cases is given in Chapter 3. Implementations of these use cases serve as scalability benchmarks for different stream processing engines and their configurations. Furthermore, Henning and Hasselbring [2020] present 7 different workload dimensions by which a system under test may or may not scale. Additionally, they provide a benchmark framework for executing benchmarks as well as analyzing and monitoring the resulting metrics. A reference implementation for the stream processing engine Kafka Streams containing the benchmark execution framework was published alongside the paper.
2. Foundations and Technologies

2.5.1 Workload Dimensions

In the definition of scalability from Section 2.1, the term load is used but not further specified. Actually there are many different dimensions that a workload can increase by, sometimes depending on the use case.

There are two workload dimensions that generally apply to stream processing. The first dimension is the message frequency which is the amount of messages per key and time. It is bounded by a single instance throughput of a stream processing engine as it primarily achieves horizontal scalability by partitioning on different keys. The second dimension is the amount of different keys. Because stream processing engines are designed to scale with increasing number of keys, this workload dimension is the more interesting one for scalability benchmarking.

Use case specific workload dimensions include the number of sensors per sensor group and the depth of nested sensor groups in hierarchical sensor aggregation, the time window size in downsampling and time attribute based sensor aggregation as well as the amount of overlapping time windows and the amount of time attribute values for the latter.

2.5.2 Benchmarking Method and Metrics

The scalability metric of Theodolite is defined as the number of least required instances per given workload. The results of applying this metric can be displayed in a scalability graph.

To identify the minimum of instances required per workload we decide for each workload if a given number of instances is able to handle given workload. This binary decision can be made by different metrics. In some cases, the stream processing engine might guarantee a certain amount of throughput and latency which can be compared to actual measured values of those metrics to determine if the workload can be handled. This method is difficult to apply because the required guarantees are not always provided and the system under test is partially involved in benchmarking itself.

A more appropriate and generally usable method is to observe messages queuing up before or while processing. When no messages are queuing up, the amount of instances were able to handle the given workload. In reality, messages might temporarily queue up due to batching or shared resources. Because of that, linear regression is applied to the function of queued up messages per time to get a trend of the queueing rate. The gradient of the trend describes the average number of messages per time unit by which the queue size increases or decreases. Furthermore, a threshold for this gradient is used to determine if the workload can or cannot be handled. The amount of queued up messages can either be retrieved from the stream processing engine or from the messaging system.

2.5.3 Execution Framework Architecture

In addition to the benchmarking method, Henning and Hasselbring [2020] provide a framework for executing scalability benchmarks. This framework includes components
2.6. The Stream Processing Framework Apache Flink

Apache Flink [Carbone et al. 2015b], formerly known as the Stratosphere platform [Alexandrov et al. 2014] before it became an Apache project, is an open-source stream processing framework developed by the Apache Software Foundation and written in Java and Scala.

The framework provides high throughput, low latency, event time processing as well as state management [Carbone et al. 2017] and fault tolerance. Apache Flink supports stream processing as well as batch processing providing high level libraries for Java, Scala, Python, and SQL. Applications are written as declarative steps that describe how the data streams or data sets should be transformed. They are then compiled and optimized before execution on a single machine, cluster, or in a cloud environment.

Flink does not provide a system for data storage and therefore relies on some other technology. For interoperability with other ecosystems, Flink provides data source and sink connectors for many other big data frameworks such as Amazon Kinesis, Apache Hadoop (HDFS), Apache Cassandra, ElasticSearch, and Apache Kafka [Flink 2020].

2.6.1 Jobs and Operators

A Flink application, so called job, consists of a set of data sources and sinks with a set of operators connecting sources to sinks. These form a directed dataflow graph that describes the Flink job. The descriptions of those dataflow graphs and operators can be declarative with functional operations like map, reduce and filter or imperative with a lower level interface for processing the data. Operators can be stateless like map and filter as well as stateful like reduce. Moreover, Flink allows the user to add custom operator state to otherwise stateless operators.

2.6.2 Parallelism

Flink achieves parallelism by data partitioning and operator pipelining in the case of non-chained operators (see Section 2.6.3). Data streams can be partitioned by a key (using keyBy) or load balanced (using rebalance or shuffle). Some data sources such as the Kafka source provide the option to run in parallel as well. Each parallel subtask of an operator then processes only parts of the stream allowing the system to be scaled up by the number of keys, and through load balancing also the by frequency of records in the data stream. In addition to a global parallelism setting that each operator in a job is configured with by
default, individual operators can have different parallelism settings. This can be useful for applications where one operator imposes a bottleneck on the whole processing pipeline and needs to be configured with a higher parallelism in order to keep up, while the other operators are still able to handle the load with a lower parallelism setting. Flink currently does not support dynamic changes of parallelism at runtime. Thus, autoscaling is difficult to achieve as a restart of the job is necessary whenever the application is rescaled. The feature for better support of autoscaling is currently in development.

### 2.6.3 Architecture of a Flink Cluster

Flink needs its own cluster for executing a stream or batch processing job. Figure 2.1 shows the architecture of a standard Flink cluster. The cluster consists of one jobmanager and one or more taskmanagers. The jobmanager controls the execution of jobs, scheduling individual tasks to the taskmanagers and handling distributed checkpointing of the application state.

---

The taskmanagers are the worker nodes which actually execute the tasks defined in the given job. Each taskmanager can offer one or more task slots which indicates the number of concurrent tasks that can run on the taskmanager. For reducing overhead of thread synchronization and buffering, Flink may chain operators together into one task. Also multiple tasks of the same job can share a task slot for better resource utilization. Because of that, a Flink cluster only needs as many task slots as the highest operator parallelism in the job and there is no need to calculate the necessary amount of task slots when deploying a job.

For a high availability setup, multiple jobmanager instances can be created with one of them being the active leader and the other jobmanagers being in standby, ready to take over the role in case that the leader fails.

### 2.6.4 Event Time Processing and Watermarks

In stream processing it is sometimes necessary to wait for more data to arrive before that data can be processed properly. The use of a window operator is the main reason for that. Generally, a window operator groups consecutive elements in a data stream together to be processed as a whole. There are different kinds of windows in Flink including time windows which group data in a time interval, count windows which always collect a specified amount of elements, session windows which group elements if their timestamps are close together and global windows that can be modified with custom triggers which indicate that a window should be closed. For many of these window operators some notion of time is necessary for them to work. Flink has two main notions of time. With processing time, a stream does not need to contain any timestamps and window operators simply take the current system time for grouping elements together. The time at which a record arrives at the window operator determines in which window that record belongs.

But if records are stored in a message broker and processed asynchronously from their creation, processing time would not achieve the desired results, especially with a large difference between creation and processing time. For this purpose the notion of event time can be utilized. Each record must contain a timestamp which indicates the records time of creation, the event time. Once a record reaches a window operator, this timestamp is used to determine in which window the record belongs. The window operator however needs a way to identify the overall progress of event time to be able to close a window after the defined time. For this reason, Flink uses watermarks which are extracted from records at or directly after the data source and indicate the progress of event time. Watermarks are part of the data stream and are inserted between records, declaring that no records with smaller timestamps than the watermark should occur in the rest of the stream.

Events arriving out of order may cause a problem in the processing of events with event time. In this case the application has to wait a certain time for records arriving late. As the waiting time can only be finite and larger values have an impact on latency, the output of an event time based streaming application might not be completely deterministic. In the case that a record still arrives at a window operator later than the current watermark,
2. Foundations and Technologies

even considering the waiting time for out of order records, it is possible to configure the maximum lateness for those records and the behaviour of the operator, for example that it should evaluate the window again and output another result. By default, the maximum lateness is zero and records are dropped when they violate the timestamp monotony of the watermark system.

In parallel streams, every parallel subtask of a data source generates independent watermarks which advance the event time of each operator that they arrive at. In case that an operator receives data from multiple different streams or stream partitions, for example through join operations, keying, or rebalancing, the lowest watermark out of all input watermarks indicates the operators current event time.

2.6.5 Fault Tolerance

Another important aspect of a distributed data processing system is fault tolerance. Flinks stream processing system achieves fault tolerance through a combination of stream replay and checkpointing. A checkpoint stores the state of every operator for a specific point in a data stream such that the stream can be reprocessed consistently with exactly-once processing guarantees from the last checkpoint in case of a failure. In addition to periodic checkpointing, Flink supports the manual creation of savepoints. A Flink job can be resumed from a savepoint such that the operator state is not lost when the job needs to be restarted. Reasons for a restart might be to rescale the job to a different level of parallelism or the modification of the job graph which is possible without losing operator state, as each operator has its own unique identifier.

For both checkpointing and savepointing, a variant of the Chandy-Lamport algorithm [Chandy and Lamport 1985], a snapshotting algorithm for distributed systems, is used. The mechanism is called asynchronous barrier snapshotting [Carbone et al. 2015a]. Checkpointing barriers are inserted into the stream, marking a point up to that elements of the stream are included in the checkpointed state. When a checkpoint is restored, the stream has to be reconsumed from the corresponding checkpointing barrier.

2.7 The Distributed Streaming Platform Apache Kafka

Apache Kafka is a scalable fault tolerant distributed system for storing and distributing streams of data. It acts as a message broker between producers and consumers that can publish and subscribe to specific data topics. The data is stored as key-value records including a timestamp in those topics as a transaction log that can be split up into multiple partitions based upon the records key. Partitioning of a topic allows for parallel processing of independent data that is records with different keys (data parallelism) [Kreps et al. 2011]. Apache Kafka will be used as the data source and sink in the implementations of use cases for Apache Flink.
2.8 The Containerization Technology Docker

Kafka Streams

Kafka Streams is a stream processing framework for building streaming microservices on top of Kafka. In Kafka Streams each instance of a stream processing application is a Java application that uses the Kafka Streams library to communicate and coordinate the execution and partitioning of the processing work via the Kafka broker. This library also allows to specify declarative steps for defining the functionality of the stream processing application. Stateful operations also use the Kafka broker as their operator state.

2.8 The Containerization Technology Docker

Docker is an open source containerization platform for Linux systems. Containerization is a lightweight virtualization technique where each container still operates through the host system but is isolated from the host and other running containers. This results in less overhead in comparison to traditional virtualization. Because containers are isolated, they contain their own execution environments for the software that runs in the container. It is therefore suited for continuous deployment purposes [Docker 2020].

Containers are created from container images (Docker images). A Docker image contains a filesystem image with the application and all of its dependencies as well as instructions on how to start the contained application. Docker images are defined by Dockerfiles which can build upon and extend base images, add files to the image, run commands, install software dependencies and initialize the application. We are going to use Docker containers for the deployment of our benchmarks.

2.9 The Container Orchestration Platform Kubernetes

Kubernetes is an open source system for container orchestration. It handles the scheduling, deployment, scaling and management of containerized applications on a cluster of multiple computing nodes and supports Docker as well as other containerisation platforms [Kubernetes 2020]. Important concepts of Kubernetes are pods, services and deployments.

A pod is an abstraction over containers that usually contains one application utilizing one or possibly more containers. From the applications point of view, a pod behaves like a node with its own hostname, address and storage.

Deployments are higher level declarative definitions of applications that manage pods automatically to achieve the defined application state by for example scaling to a higher number of pods or restarting failed pods or containers.

A service is an abstraction layer for network communication to and from Kubernetes deployed applications which has one fixed hostname and address to abstract away the possibly many pods with different and changing addresses due to pods being considered disposable. The service also defines all ports that the application listens on.
2. Foundations and Technologies

Kubernetes will be essential for the scalability benchmarks because it enables us to easily scale up the system that is being benchmarked.

2.10 The Time Series Database Prometheus

Prometheus is a time series database (TSDB) combined with a monitoring and alerting toolkit [Prometheus 2020]. The TSDB of Prometheus is a non-relational database optimized for storing and querying of metrics as time series and therefore can be classified as NoSQL. The Prometheus database and toolkit has the ability to be deployed at scale by duplication of the service. This scaled deployment allows for highly available monitoring applications as well as scalability by the amount of monitoring targets using data sharding.

Prometheus mainly uses the pull method where every application that should be monitored has to expose a metrics endpoint in the form of an HTTP-REST-API either by the application itself or by a metrics exporter application running alongside the monitored application. The monitoring toolkit then pulls the metrics data from those endpoints in a specified interval.

Prometheus uses its own querying language called PromQL which is better optimized than SQL for defining queries on time series data. We are going to use Prometheus for monitoring of the system under test, that is Flink and Kafka and collecting metrics for later analysis while running benchmarks.

2.11 The Visualization Dashboard Grafana

Grafana is a flexible dashboard for visualizing monitoring and analysis data [Grafana 2020]. It features multiple different visualization types and graphs and has many configuration options for customizing and adapting to a desired use case. Grafana has many plugins for extending the dashboards panels, adding data sources and even controlling applications through the dashboard. Many data sources are included in the official Grafana plugins such as the Prometheus data source. We are going to use Grafana with the Prometheus data source for live monitoring while running benchmarks.
In this chapter we will describe each of the use cases in detail, discuss changes to the architecture that we had to make and explain the deployment of the applications and the Flink cluster to Docker and Kubernetes. Our Flink implementation of the four use cases from Theodolite [Henning and Hasselbring 2020] is based upon the reference implementation in Kafka Streams. Because of some major differences between the two stream processing frameworks, we need to change large parts of the implementation, especially for the hierarchical sensor aggregation. The implementation is publicly available on GitHub\(^1\).

### 3.1 Considerations on the Integration of Flink with Kafka

Flink has connectors for integration with many other systems in the big data ecosystem such as Kafka. To make use of this connector, we need to specify how to serialize and deserialize data to and from Kafka topics.

Another important aspect to consider is that the parallelism \(n\) of Flink and the partitions \(p\) of a Kafka topic can be different. In the case that \(n = p\), each Flink instance simply handles one Kafka partition. But for certain applications the Kafka broker might be able to handle the load with \(p\) partitions but the processing is more computationally expensive and needs more instances \(n > p\) to handle the load.

In that case Flink forwards the data from the \(p\) partitions only to \(p\) Flink instances while \(n - p\) instances are idling. Also, if there are more Kafka partitions than Flink instances \(n < p\) which might occur in less computationally complex applications, the data of \(p - n\) partitions will not be processed.

To make use of all \(n\) instances or process all \(p\) partitions respectively, we need to balance the load equally after consuming the data from the Kafka topic. There are two common ways of load balancing between Flink operators. The `shuffle` operation generates a pseudo random number for each record that decides which instance will be responsible for the processing of that record. Furthermore, the `rebalance` operation distributes each arriving record to the next operator in a round-robin fashion. Because pseudo random number generation takes a significant amount of time and might also not be the fairest method of balancing, we will apply a `rebalance` operation directly after consuming from a Kafka topic.

\(^1\)https://github.com/NicoBiernat/theodolite
3. Approach

3.2 Serialization

As already mentioned, we need to specify the serialization and deserialization schema for reading from and writing to Kafka topics. Furthermore, Flink has a sophisticated type and serialization system which can handle primitive Types, Flink tuples, and Java objects which fulfill certain criteria very efficiently. But when dealing with objects that do not comply with those criteria, we need to specify serializers and deserializers and register them at the internally used serialization framework Kryo\(^2\) to prevent performance loss due to serialization overhead emerging from Kryos fallback serializer [Kruber 2020b].

Because we need to integrate our implementation with the workload generators, we have to deal with types from the Kafka Streams implementation which do not comply with Flunks criteria for efficient serialization. Examples of such types are the record types `ActivePowerRecord` and `AggregatedActivePowerRecord`. In the following we are going to explore possible solutions to achieve efficient serialization of those types.

One solution might be to adjust the classes of the Java objects that are processed by Flink such that they comply with Flinks criteria. This would require the modification of library code including the addition of public constructors and public getter and setter methods for all fields. Another option would be to avoid using those data objects inside of Flink and instead use Flink tuples which can be serialized much faster. Although this would most likely be the fastest option, the additional functionality that some of those objects contain needs to be rewritten to work on tuples.

Flink also supports serialization using Apache Avro\(^3\) for internal use as well as for Kafka serialization. Avro provides the possibility of schema evolution while delivering reasonable performance and is probably the best overall option especially when schema evolution is needed to stay future proof. But the migration to Avro would entail modifications in the workload generators and the whole serialization part of the use case implementations.

The last option is to provide serializers and deserializers for each of those objects and register them to be used by the Kryo serializer. Fortunately it is possible to wrap the serializers and deserializers from the Kafka Streams implementation into the required interfaces. This option is the easiest to accomplish while providing a performance improvement comparable to Flinks builtin object serialization. We draw this conclusion because for every type we provide custom serializers which we expect to be as performant as other serialization frameworks registered via Kryo that were benchmarked by Kruber [2020b]. Note that the serializer and deserializer implementations themself need to be serializable for Flink to transfer them over the network. Moreover, we are able to combine the Kafka and Kryo serialization for the two mentioned record types.

\(^2\)https://github.com/EsotericSoftware/kryo
\(^3\)http://avro.apache.org/
3.3 Use Case 1: Database Storage

The first use case reads records from a Kafka source, transforms each record into a database specific format and then stores it in a database. This streaming topology is shown in Figure 3.1. The performance behaviour of this use case is tightly coupled with that of the underlying database system.

In comparison to the much simpler format transformation, the database operation would be the bottleneck of the application. Because of that, we would primarily measure the write throughput of the database instead of the stream processing frameworks scalability. That is why we follow the decision of Henning and Hasselbring [2020] to omit the database completely and instead print each record to the standard output.

Listing 3.1. Implementation of Use Case 1 in Flink

```java
1 FlinkMonitoringRecordSerde<ActivePowerRecord, ActivePowerRecordFactory> serde =
2     new FlinkMonitoringRecordSerde<>(inputTopic, ActivePowerRecord.class,
3         ActivePowerRecordFactory.class);
4 FlinkKafkaConsumer<ActivePowerRecord> kafkaConsumer =
5     new FlinkKafkaConsumer<>(inputTopic, serde, kafkaProps);
6 kafkaConsumer.setStartFromGroupOffsets();
7 kafkaConsumer.setCommitOffsetsOnCheckpoints(true);
8 StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
9 env.enableCheckpointing(commitIntervalMs);
10 StreamExecutionEnvironment env = env.addSource(kafkaConsumer);
11 .rebalance()
12 .map(v -> "ActivePowerRecord{\n13     + "identifier:" + v.getIdentifier() + "\n14     + "timestamp:" + v.getTimestamp() + "\n15     + "valueInW:" + v.getValueInW() + "\n16 .print();
17 env.execute(applicationId);
```

Listing 3.1 shows the stream processing job of the modified use case 1 in Java code (shortened). First, required configurations are loaded from environment variables. After that, a Serde (combined serializer and deserializer) is instantiated.

The used FlinkMonitoringRecordSerde is a serializable, Flink compliant wrapper of the already existing Serde from the Kafka Streams implementation. A new FlinkKafkaConsumer is created which connects to the Kafka broker and also participates in Flink’s checkpointing for fault tolerance and therefore guarantees exactly once processing of records. It can create underlying parallel Kafka Consumers for load balancing. We set the KafkaConsumer to start reading from the consumer group offset, which is unique per application and stored by
3. Approach

Kafka, as well as commit those offsets to Kafka on each checkpoint.

Next, we get an instance of a `StreamExecutionEnvironment` on which we can then define our stream processing topology and activate checkpointing. A new `DataStream` is created by adding the `FlinkKafkaConsumer` as a new source to the `StreamExecutionEnvironment`. We then transform each record which was deserialized into an `ActivePowerRecord` to a `String` using a map operator and print the result to the standard output. However, when running this application on multiple nodes, the record is only printed on the node that was responsible for the processing of that record.

3.4 Use Case 2: Hierarchical Aggregation

The second use case is based on the hierarchical aggregation of sensor data [Henning and Hasselbring 2019]. This is by far the most complex use case out of the four. Because of this, we need to make a lot of modifications to transfer the application from the Kafka Streams model to Flinks stream processing model.

Figure 3.2 shows the original streaming topology of the use case from the Kafka Streams implementation. The application receives data from three input topics. In the sensor data topic all measurements from physical sensors arrive. The sensor config topic receives updates to the sensor hierarchy such that the application can dynamically change its configuration. And the aggregation result topic acts as a feedback loop from the output in order to further aggregate aggregated sensors, when specified in the hierarchy.

Incoming sensor records are then merged with the aggregated records which requires a previous mapping operation on the aggregated records that is omitted in the figure. Also omitted is the preprocessing of the sensor hierarchy into a stream of key-value pairs containing each sensor with a list of the aggregated sensor groups that it belongs to. The record stream is then joined together with the sensor hierarchy which yields a stream containing each sensor together with its value and the list of sensor groups. Those records are then duplicated for each entry in the sensor group list such that a record with a combined key of sensor and group contains the value for the sensor.

The resulting stream is transformed into a table representation to make sure that previous sensor values are available for the following aggregation if some of the required records arrive late. The records are grouped by their combined key and aggregated over consecutive time windows, so called tumbling time windows, which are shown in Figure 3.5.
3.4. Use Case 2: Hierarchical Aggregation

The time windows are not shown in Figure 3.2.

Large parts of the streaming topology, including the join and duplicate operations as well as the last values table, are based on the Dual Streaming Model which allows data streams to be interpreted as continuously updating tables [Sax et al. 2018]. Those tables can then be converted back into changelog streams. However, Flink does not include this feature in the same way as Kafka Streams. While it is possible to convert data streams into tables, execute continuous queries using SQL and convert the tables back into data streams, the conversion is less flexible, more difficult to use and probably incorporates a performance loss because additionally to the Stream API we would have to use Flinks Table API. Although Flinks new Table Planner can reach nearly the performance of pure stream processing [Kruber 2020a], this certainly is not the idiomatic way of porting this use case to Flink.

However unlike Kafka Streams, Flink supports quite powerful low level stateful streaming operations which we can use to achieve the same functionality without the need for the high level Dual Streaming Model.

In Figure 3.3 we present the modified streaming topology using Flinks low level stateful streaming operators.

Instead of joining the merged input stream with the configuration stream into a table and then duplicating the records, we connect the two streams which allows us to define the following operation on both streams simultaneously and use a shared state in that operation. We then key both streams by the sensor identifier such that each parallel subtask of the following operator manages only parts of the incoming records as well as sensor configuration state. Next, we apply a CoFlatMapFunction which allows us to specify two regular flatmap functions, one for each of the two streams. Both of these functions have access to the shared operator state. The function that handles the configuration stream saves incoming configurations into the operators MapState. It also emits a null record to indicate when a sensor was deleted from the configuration. The other function which
3. Approach

is responsible for processing the incoming sensor records then reads the corresponding configuration saved in the shared MapState and outputs a new record for every entry in the list of groups that the sensor belongs to. The emitted record has a combined key of sensor identifier and group identifier.

After the duplication, we do not save the last aggregation results in a table. Instead we first keyBy the sensor group and open a tumbling time window for each group. The last aggregation value is then stored inside of the ProcessWindowFunctions operator state together with the last values of all sensors that belong to the aggregations sensor group. This ProcessWindowFunction aggregates all records for each sensor group in a window using the last values when expected records are missing or arriving late and outputs exactly one aggregated value per group key and window. The aggregation results are then written back to the Kafka output topic which also feeds back into the input stream of the application to allow for the aggregation of aggregated sensor records.

Figure 3.3. Streaming Topology of Use Case 2 for Flink
3.5 Use Case 3: Downsampling

Downsampling is the process of reducing the number of records in a stream of data per time frame. It is very useful for applications which need data from a topic with high velocity but cannot handle the amount and frequency of records. In that case a downsampling service could aggregate the records in a meaningful way and produce aggregated records in slower and more regular intervals.

Figure 3.4 shows the dataflow graph for downsampling. We apply a keyBy which groups the input stream by a specified key, in this case the sensor identifier, and then calculate a statistical aggregate using time windowed aggregation with non-overlapping tumbling windows that can be configured in minutes as shown in Figure 3.5.

For the aggregation operator, an AggregateFunction needs to be implemented according to an interface such that Flink can carry out incremental aggregation on the data. Alternatively a ProcessWindowFunction can be defined which runs on all records after the window is closed. This more general approach is very inefficient compared to the incremental aggregation of an AggregateFunction but on the other hand the ProcessWindowFunction provides additional information about the window and the output.

Fortunately it is possible to combine both approaches and do an incremental aggregation with the AggregateFunction first and then define a ProcessWindowFunction which runs on the single aggregation result but can make use of the context information. This is used to add the key back to the output which was lost during aggregation because it is not part of the aggregate state. The result is emitted as a Tuple2<String, Stats> where the first element is the key and the second element is the statistical aggregation result. It is then transformed into a String, logged to standard output and written back to Kafka using the FlinkKafkaKeyValueSerde.

This Serde is similar to FlinkMonitoringRecordSerde a wrapper for the Kafka Streams Serde but is additionally able to serialize and deserialize the key, value and timestamp of the Kafka records instead of just the value. This was not necessary in deserialization because the records key is redundantly stored inside of the value as the sensor identifier. This specific functionality though is required if a key-value pair needs to be written back to Kafka.

---

3. Approach

3.6 Use Case 4: Aggregation Based on Time Attributes

The last stream processing use case involves another form of aggregation. Rather than aggregating sensor records of different sensors in a hierarchical way like in use case 2, we now aggregate the records of each sensor by a specific time attribute. Common time attributes include the hour of day and the day of week containing a record's timestamp. The collection of such data provides valuable insights on the statistics of the sensor data.

We perform the aggregation of records for every hour of the day for this use case by applying sliding time windows similar to the Kafka Streams implementation although sliding windows are called hopping windows in Kafka Streams. Unlike a tumbling window, a new sliding window can start before the previous window is closed. Hence, multiple windows can be active at the same time with overlapping time ranges. Figure 3.7 shows an example of sliding windows. The window is specified by its size and its slide. The slide value is the time after which a new window is opened. So a tumbling window is the same as a sliding window with equal size and slide parameters.

The incentive is that we want to aggregate the values for each hour of day over the duration of a week or month while not always strictly starting the aggregation on the first
3.6. Use Case 4: Aggregation Based on Time Attributes

![Streaming Topology of Use Case 4](image)

**Figure 3.6.** Streaming Topology of Use Case 4 [Henning and Hasselbring 2020]

...day of the week or month but rather starting on every day. This allows us to later retrieve the aggregated value for a week or month from any starting day. Due to the many and rather long-timed time windows, this use case is likely to be the most computationally and memory intensive out of the four use cases.

---

3. Approach

3.7 Kubernetes Deployment for Use Case Implementations with Flink

3.7.1 Selecting a Suitable Deployment Type for Flink

Flink supports many deployment targets including local clusters for development, standalone clusters for bare-metal servers or virtual machines, as well as many resource managers such as YARN, Mesos, Docker and Kubernetes. A Flink cluster can be deployed in different deployment modes.

In session mode, one Flink cluster is deployed where multiple jobs can be submitted and executed. This has the advantage of avoiding the overhead of creating one cluster for each job but the disadvantage that jobs are not isolated from each other. If one job fails and takes down the taskmanagers, all other jobs running on that task manager will be affected and need to recover. Additionally, the load on the jobmanager will increase due to the book-keeping of all jobs that are running in the cluster. Also, this deployment method does not easily comply with the Docker and Kubernetes deployment methods where the whole application is packaged into a container image. Instead we would have to deploy the Flink cluster and then submit the use case job separately.

In Per-Job mode a new Flink cluster is created for every job. This mode comes with more overhead when running more than one Flink job, but since we are going to execute one benchmark at a time, the overhead is not relevant for our deployment. The Per-Job mode is recommended for production deployments because of the isolation between jobs regarding failing jobs and their recovery. Also, the Per-Job mode is easier to use for our use case implementations as we can package the whole use case application as one Docker image. This image contains the Flink cluster and the use case job and can be started either as a jobmanager or a taskmanager.

3.7.2 Deployment of a Flink Job Cluster to Kubernetes

For the deployment of a job cluster to Kubernetes we need to specify multiple Kubernetes resources. First, we provide separate Kubernetes deployments for the jobmanager and the taskmanager. For the jobmanager we actually use a Kubernetes job which is similar to a normal deployment but might finish execution at some point and terminate. As we do not need a high-availability setup, only the taskmanager deployment is going to be scaled up. On both deployments, we need to expose the ports that Flink uses for remote procedure calls and exporting metrics, on the jobmanager the ports for the object cache server (blob server) and web user interface, and on the taskmanager also for the queryable state. We further define the CPU and memory limits in both deployments and the number of replicas in the taskmanager deployment. As we are using the same container image for both the jobmanager and taskmanager, we need to define which one to start via the program arguments and specify the path to the Flink job to execute when starting the
3.8 Identification of Suitable Metrics for Benchmarking the Scalability of Flink

The configurations for Flink are provided through a ConfigMap which is a resource that contains configuration files and is mounted into the application container as a read only volume. Because Flink's startup script is editing its own configuration file in order to load configurations optionally from environment variables, we need to copy the configuration file from the ConfigMap to a different volume which allows write access for the Flink container. To achieve this, we specify an Init-Container that runs before the application container and copies the ConfigMap to a new volume with write access. Important configurations include the default parallelism, that we are using for all Flink operators, the number of task slots per taskmanager, the memory limit for the jobmanager and taskmanagers, all of the previously exposed ports as well as the Prometheus metrics reporter which exports the Flink internal metrics through a Prometheus compliant API. For the pods to be able to communicate, we define Kubernetes services for the jobmanager and taskmanager including all of the exposed ports.

3.7.3 Docker Images for Flink

Since version 1.11, Flink provides base images that contain the binaries needed to run a cluster. We can build our own Docker images on top of those images by simply adding the Flink job. Unfortunately, Flink does not provide a base image for Java versions other than Java 8, which is why we modified the official Dockerfile and built our own Flink base image that we can use with Java 11. All Docker images that we used are publicly available on Dockerhub. 6

3.8 Identification of Suitable Metrics for Benchmarking the Scalability of Flink

For applying the Theodolite benchmarking method we need to identify metrics that can be used directly or indirectly as a measurement for scalability. In Section 2.5.2 we described that observing the speed at which records are queueing up in the message queue or in the stream processing engine can be used together with a threshold to indicate if the system can handle the given workload. The metric that describes the number of records in the message queue is usually called consumer-lag or record-lag. We first tried to use the Kafka-Lag-Exporter 7, a metrics exporter from Kafka to Prometheus, which was used by Henning and Hasselbring [2020]. But because of an issue with Flink not committing its record offsets back to Kafka in a way that this exporter can handle, we used a metric that Flink provides itself.

---

6 https://hub.docker.com/u/nicobiernat
7 https://github.com/lightbend/kafka-lag-exporter
3. Approach

The Flink metric did not provide results as stable and reliable as the Kafka-Lag-Exporter. Sometimes, metrics are exported late or not accurately enough. The problem with metrics arriving too late could be solved by increasing the export interval but the values were not reliable enough especially when the Flink cluster was under heavy load. So for benchmarking scalability, a system under test should rather be monitored by an external application to avoid such problems and provide consistent and comparable results. Fortunately, an update of the Kafka-Lag-Exporter solved the issue we faced earlier. Hence, we were able to use the Kafka-Lag-Exporter instead to monitor the record lag of the Flink application.

3.9 Integration with the Theodolite Execution Framework

The architecture of the Theodolite benchmark execution framework is shown in Figure 3.8. It is designed to run almost completely in a Kubernetes cluster, except for the experiment control and offline analysis. In the following, we are going to describe the parts of the architecture in more detail. The experiment control consists of a set of scripts that control the deployment, configuration, and execution of benchmark components as well as the persistent storage of results after each benchmark execution for later analysis.

The parameters that need to be specified are the use case to run, which workload dimensions and instance values to benchmark, as well as parts of the stream processing
3.9. Integration with the Theodolite Execution Framework

ingines and messaging systems configuration. This configuration includes the number of partitions for Kafka, resource limitations (CPU and Memory) for the stream processing engine which are enforced by Kubernetes per instance, and the Kafka commit interval that specifies how often the application will commit its current consumer offset back to the Kafka broker. In the case of Flink, we changed this configuration to be the checkpointing interval because Flink automatically commits its offsets on every checkpoint. The last parameter specifies how long each benchmark should be executed.

The experiment control is executed on a controller node or a local machine where it remotely connects to a the Kubernetes cluster in a cloud environment and sets up the necessary components. First, the Kafka cluster is configured and all topics for the benchmark are created with the specified number of partitions. Then the workload generator for the chosen use case is started with the configured workload dimension. In the case that the workload dimension is higher than one workload generators output capability, the workload generator is automatically scaled up to be able to deliver the requested workload.

After that, the stream processing engine, in this case Flink, is deployed together with the streaming job that implements the specified use case and its configuration. This includes Kubernetes and Flink resource limits, the number of instances, as well as Flink specific configuration such as the number of task slots per taskmanager. The experiment control then waits a given number of minutes for the benchmark to finish.

While the benchmarks execute, Kafka and Flink expose metrics to be collected by the monitoring component. The monitoring component consists of a Prometheus database that periodically pulls metrics from the configured metrics endpoints and stores them for visualization and later analysis. The Grafana dashboard is set up to continuously query and visualize those metrics for live monitoring of the benchmark execution.

When the benchmark finishes, the experiment control pulls all relevant metrics about the benchmark execution from the Prometheus database and saves it locally for later analysis. It then stops the stream processing engine and workload generator and deletes the Kafka topics so that the following benchmark can be executed without any remaining and possibly conflicting data in the messaging system.
Chapter 4

Evaluation

In this chapter we describe how we execute the scalability benchmarks for Flink, including our experiment setup, the used hardware and software, our methodology which is based on the Theodolite benchmarking method, as well as the test cases that we want to evaluate. We then continue by presenting and discussing our results and considering possible threats to validity.

4.1 Experiment Setup

For executing the scalability benchmarks we are using a cloud environment which can easily be set up and reproduced for comparability to results from other scalability benchmarking work. The cloud provider that we are using is the Oracle cloud and we are specifically using the managed Kubernetes cluster with one node pool. Because of resource limitations, the maximum number of nodes that we could assign to that cluster is bounded to 9 nodes. Table 4.1 shows the specification of the virtual machine nodes that we are using for the Kubernetes cluster.

For the software setup, we use the confluent platform that includes the Kafka broker and Zookeeper which is used by Kafka. Additionally, we use the Kafka-Lag-Exporter for exporting metrics about the Kafka broker, consumer offsets, and record lag. Moreover, we deploy Prometheus through the Prometheus-Operator for collecting and providing metrics and Grafana for visualizing them. Finally, we use Flink as the system under test. The versions of the software that we are using is shown in Table 4.2. We are using 10 Kafka instances and 3 Zookeeper instances throughout all benchmark runs, to ensure that the message broker is not imposing a bottleneck on the system.

Table 4.1. Kubernetes Cluster for Executing Scalability Benchmarks

<table>
<thead>
<tr>
<th>Kubernetes Cluster</th>
<th>Version 1.16.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes (VM.Standard.E2.2)</td>
<td>9</td>
</tr>
<tr>
<td>CPU</td>
<td>AMD EPYC 7551</td>
</tr>
<tr>
<td></td>
<td>2.0GHz - 3.0GHz</td>
</tr>
<tr>
<td>CPU Cores per Node</td>
<td>2 Cores / 4 Threads</td>
</tr>
<tr>
<td>Memory per Node</td>
<td>16GB</td>
</tr>
</tbody>
</table>
4. Evaluation

Table 4.2. Software Versions

<table>
<thead>
<tr>
<th>Software</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confluent Platform</td>
<td>5.5.0</td>
</tr>
<tr>
<td>Kafka-Lag-Exporter</td>
<td>0.6.3</td>
</tr>
<tr>
<td>Prometheus Operator</td>
<td>8.13.8</td>
</tr>
<tr>
<td>Grafana</td>
<td>7.1.1</td>
</tr>
<tr>
<td>Flink</td>
<td>1.11</td>
</tr>
<tr>
<td>Java Runtime (for Flink Jobs)</td>
<td>11</td>
</tr>
</tbody>
</table>

4.2 Methodology

We largely adopt the benchmark execution and evaluation methodology from Theodolite [Henning and Hasselbring 2020] as described in Section 2.5.2.

We are going to benchmark each test case for a specific use case with a set of workload values and instances where sub-benchmarks are executed for every possible combination of workload value and number of instances. The sub-benchmark runs of a test case are monitored by Prometheus while executing and the resulting metrics of the record lag are retrieved and stored locally after every run. The raw record lag data of each of the sub-benchmark executions is analyzed using linear regression to obtain the trend slope of the number of queued messages. When this slope is below a certain threshold value of 2000 (this value was adopted from Theodolite), we conclude that the workload could be handled by the number of instances. We then find for every tested workload value, the lowest number of instances that could handle that workload. From these results we can construct a scalability graph where the lowest number of instances needed (Y-axis) is dependent on the workload (X-axis).

4.3 Test Cases

Our goal is to evaluate the effect of different configurations on the scalability of Flink. Therefore, we need to identify test cases that might have an impact on the scalability of the use cases. Flink provides a large number of configuration options. Hence, we need to focus on inspecting some important ones. For all other configurations, Flink provides default values that are suitable for most applications. We decided that the checkpointing interval might be an interesting option to test, as a higher fault tolerance could have an impact on scalability. Also, we want to investigate how the scalability behaves when we provide each taskmanager with two instead of only one taskslot. We assume that we would need half as many instances to handle a given workload as we could achieve a higher parallelism with less instances but that might not actually be the case. And lastly, we choose to compare different state backends that are used in the stateful operators of use cases 2, 3, and 4. We are going to test the default in-memory state backend as well as the
4.4. Results and Discussion

4.4.1 Use Case 1: Database Storage

Figure 4.1 shows the scalability of the first use case with different numbers of taskslots per taskmanager. Clearly, the second taskslot seems to improve the scalability but unlike we expected, it does not result in always half the number of instances being necessary in comparison to one taskslot. At two points, namely 300000 and 500000 data sources, exactly the same number of instances are needed regardless of the number of taskslots. For the rest of the tested workloads, the configuration with two taskslots needs two instances less than the configuration with just one.

We compare different checkpointing intervals and how they affect the scalability in Figure 4.2. Our observations show that the shorter interval of 100ms negatively impacts the scalability compared to a longer interval of 10s as two more instances were needed for most of the workloads with shorter checkpointing. Moreover, we encountered an anomaly in the results as a workload of 500000 data sources could only be handled by 6 instances, but 600000 data sources could be handled by 4 instances. This result should, in theory, not be possible as a scalability graph is always increasing monotonously. A closer look into the recorded data shows that the number of queued records is constantly rising and falling

---

1https://rocksdb.org/
4. Evaluation

![Scalability graph of use case 1 with different checkpointing intervals](image)

**Figure 4.2.** Scalability graph of use case 1 with different checkpointing intervals

![Scalability graph of use case 2 with different numbers of taskslots per taskmanager](image)

**Figure 4.3.** Scalability graph of use case 2 with different numbers of taskslots per taskmanager

in both cases. Our analysis method uses linear regression to fit a trend line to that data which might be problematic in those cases with highly fluctuating data points in which the trend can not precisely be estimated and factors such as a shifted time window of the measurement could influence the trendline and lead to false conclusions.

### 4.4.2 Use Case 2: Hierarchical Sensor Aggregation

Figure 4.3 shows the scalability of the second use case with different numbers of taskslots per taskmanager. Note that the workload dimension for this use case is the number of nested groups in the sensor hierarchy instead of the overall number of data sources. Unlike the number of data sources, this workload does not follow a linear scale, but an exponential one instead as the number of sensors and therefore the amount of incoming data grows exponentially with the depth of the sensor hierarchy. With one taskslot, only workloads up to 8 nested groups could be handled by 4 instances and 9 nested groups could not be
4.4. Results and Discussion

The scalability of the hierarchical sensor aggregation with different checkpointing intervals is displayed in Figure 4.4. Both graphs are completely identical, meaning that an extremely short checkpointing interval of only 100ms does not affect the scalability compared to a more relaxed checkpointing interval of 10s.

In Figure 4.5 we compare the in-memory state backend to the RocksDB state backend. We can see that the RocksDB backend scales better with this use case as we were able to process 8 nested groups with only 2 instead of 4 instances and even 9 nested groups with only 6 instances. The results are similar to using 2 taskslots per taskmanager instance. So the choice of the state backend has a comparable impact on the scalability of this use case.

handled at all by the number of instances that we tested with. When using two taskslots per instance, we were able to process 8 nested groups with only 1 instance and even 9 nested groups although 8 instances were necessary for that. We conclude that increasing the number of taskslots actually reduces the number of required instances for this use case as we expected.
4. Evaluation

![Graph showing scalability with different numbers of taskslots per taskmanager]

**Figure 4.6.** Scalability graph of use case 3 with different numbers of taskslots per taskmanager

![Graph showing scalability with different checkpointing intervals]

**Figure 4.7.** Scalability graph of use case 3 with different checkpointing intervals

as the number of taskslots per instance.

### 4.4.3 Use Case 3: Downsampling

In Figure 4.3 the scalability of the third use case is shown for different numbers of taskslots per taskmanager instance. We observe that for this use case we achieve better scalability with less taskslots per instance. This result is very unexpected as we would assume that less instances would be necessary when each of them has more taskslots. One possible explanation for this behaviour could be that the resource limit of 2 CPUs is reached with 2 taskslots and any additional overhead of the taskmanager such as checkpointing directly impacts the performance and therefore the scalability as well. With only one taskslot, the taskmanagers have some CPU resources left for any additional overhead. With this explanation, we would expect to observe similar behaviour for the other use cases which was not the case, so this behaviour seems to be use case dependent.
4.4. Results and Discussion

We continue with the comparison of different checkpointing intervals in Figure 4.7. The result of the benchmark with a checkpointing interval of 10s contains an anomaly at the workload values of 150000 and 200000 data sources. For 150000 data sources, 4 taskmanager instances were necessary whereas 200000 data sources could be handled by only 2 instances. The issue seems to be the same as described in Section 4.4.1.

Regardless of the anomaly in the data, we can see that the scalability seems to be negatively impacted by a longer checkpointing interval but we cannot draw this conclusion with confidence as the two graphs approach each other again at higher workload values of 200000 data sources and more.

Figure 4.8 shows the effect of different state backends on the scalability of use case 3. The anomaly in the RocksDB benchmark with 50000 data sources was the consequence of data loss of the results for the smaller instance values due to a crash of Prometheus during execution.

Contrary to the results of use case 2, we do not observe an improvement with RocksDB over the in-memory state backend. In fact, we can see that using the RocksDB state backend with this use case had a negative effect on scalability as a multiple of the instances was needed in order to process the workloads. The state that this use case manages is rather small compared to use case 2 and 4, so the RocksDB state backends overhead of management and disk storage probably outweighs its benefits.

4.4.4 Use Case 4: Aggregation Based on Time Attributes

In Figure 4.9 the scalability graph of the fourth use case is depicted for different numbers of taskslots. The amount of instances needed with two taskslots per instance was almost half than with one taskslot as we expected.

We can see the comparison of different checkpointing intervals for the fourth use case in Figure 4.10. Once again, our results contain an anomaly as for 3000 data sources 6 instances
4. Evaluation

Figure 4.9. Scalability graph of use case 4 with different numbers of taskslots per taskmanager

Figure 4.10. Scalability graph of use case 4 with different checkpointing intervals

were necessary whereas 4000 data sources could be handled by 4 instances. Despite that, with the 10s checkpointing interval only 2 instances were needed for 2000 data sources instead of 4 instances. For higher workloads, namely 5000 and 6000 data sources, the scalability was not affected by changing the checkpointing interval.

Finally, we are going to take a look at the impact of changing the state backend for use case 4 to RocksDB in Figure 4.11. As use case 4 needs to manage larger state because of the large and overlapping time windows, we would expect that changing the state backend may affect the overall performance of this use case significantly. And in fact the results meet our expectations in that changing to the RocksDB state backend caused this use case to need far less instances than with the in-memory state backend. For some workloads only half of the instances were needed and in other cases only a quarter of the instances. Therefore, changing the state backend to RocksDB for use cases with big state improves the scalability significantly.
4.5. Threats to Validity

In our evaluation we encountered anomalies in some results where a smaller load could only be handled by a certain number of instances but a higher load could be handled by less instances. Due to this being a contradiction, we suspect that our execution or evaluation methodology might have had an influence on the results. The analysis of the record lag data with linear regression is just an approximation of the rate at which records are queued. This approximation could lead to false negatives, meaning that a number of instances is identified as insufficient to handle a workload, although the records were only temporarily queuing up. This problem is amplified when only small datasets are used for linear regression as the overall trend might not be apparent from only a few samples. This led us to the assumption that the duration of the sub-benchmarks might be too small for an accurate analysis of the record lag behaviour.

Furthermore, we only executed each experiment once and therefore cannot guarantee that the results are free from any individual fluctuations. Also, the benchmarks were only executed only on one Kubernetes cluster that had limited resources. Although we carefully monitored the clusters resource utilization, the possibility of an influence on the results due to scheduling or insufficient resources can not be eliminated completely. Finally, the experiments could only be executed up to a limited and rather small scale of a maximum of 8 instances. Because of this, our results and conclusions might not be generalizable for higher scaled clusters and workloads.

Figure 4.11. Scalability graph of use case 4 with different state backends
Chapter 5

Related Work

The development of stream processing architectures, analysis of software scalability in cloud environments, and benchmarking of big data frameworks have gained a lot of attention and lots of research has been done in these fields. Many of the work related to this thesis focuses on the analysis of performance of stream processing engines by measuring throughput and latency. Benchmarking specifically the scalability of stream processing engines was only performed previously by Henning and Hasselbring [2020] using Kafka Streams on whose work this thesis is based upon.

Karimov et al. [2018] benchmark the throughput and latency of multiple stream processing engines, namely Storm, Spark, and Flink. They also employ use case based benchmarks and workloads from real-life applications of data analysis in the online-gaming industry. Their use cases mainly focus on windowed operators as well. The execution is also performed in a realistic environment and evaluated together with the message queue. In their evaluation they then separate the system under test from the rest of the benchmarking environment. Furthermore they propose a benchmarking framework for the metric of sustainable throughput which they defined as well. Their definition of sustainable throughput is very similar to our methodology of measuring scalability. Chintapalli et al. [2016] also benchmark throughput and latency for those three stream processing engines using different use cases but with a similar benchmark execution environment as well as Shahverdi et al. [2019] who also include two recently emerging stream processing engines, namely Kafka Streams and Hazelcast Jet, in their benchmarks.

Hanif et al. [2019] propose a benchmarking tool for stream processing engines based on the linear road benchmark [Arasu et al. 2004] which simulates a traffic environment with virtual vehicles reporting some data that has to be analyzed. They provide an implementation for Flink as well and evaluate their benchmarking results using this method regarding throughput.

Verbitskiy et al. [2016] compare distributed stream processing to serial algorithms in order to determine when it is sensible to use a distributed data processing system and when such a system might not be suitable and instead is outperformed by a more traditional single thread implementation.

One of the first benchmarking frameworks for stream processing called StreamBench was proposed by Lu et al. [2014]. It includes 7 application benchmarks from typical stream processing scenarios and addresses the performance of the system under different workload scales while taking other aspects of stream processing engines such as fault tolerance into
5. Related Work

account. For this, they introduce 4 workload suites that cover these requirements. They provide benchmark results using their framework for Storm and Spark, focussing on throughput and latency.
Chapter 6

Conclusions and Future Work

6.1 Conclusions

In Chapter 1 we defined three goals for this thesis. The first goal G1 was to implement the four use cases for stream processing that were defined by Henning and Hasselbring [2020]. We provided implementations for each use case and discussed important considerations such as the performance of serialization as well as the consequences and problems with different levels of parallelism for the message broker Kafka and the stream processing engine Flink. While most use cases could easily be transferred to Flink, most of the hierarchical sensor aggregation (use case 2) had to be modified extensively to run on Flink's stream processing architecture.

Our second goal G2 was to instrument the use case implementations from G1 as benchmarks for scalability using the Theodolite benchmarking method and framework. We achieved this goal by providing a suitable deployment for our use case implementations to Kubernetes such that they can be executed at scale in a cloud environment. We also modified some components such as the experiment control of the Theodolite benchmarking framework in order to deploy and execute our implementation as scalability benchmarks. Furthermore, we identified suitable metrics that can be used for monitoring of the number of queued up records before the stream processing application such that we can derive results about whether the tested configuration was able to handle a given workload.

The final goal G3 was to execute our scalability benchmarks for different configurations of Flink and evaluate the results. In Chapter 4 we described our benchmarking environment in a public cloud and the methodology employed for benchmarking and analyzing the results. We then executed multiple benchmarks for the four use cases with different configurations of Flink that we identified before, which are the checkpointing interval, the number of taskslots per taskmanager and the state backend. The results showed that checkpointing usually does not affect scalability much although in some easier to compute use cases, for example in our first use case, it might not be advisable to set the interval too low as it showed a negative effect on the scalability. A higher number of taskslots per taskmanager affected the scalability positively for most use cases as we expected because every taskmanager instance could execute multiple computations in parallel through the use of multithreading. For the third use case though, more taskslots had a negative effect on scalability. So the benefit of multiple taskslots seems to be use case dependent as...
the resources of the taskmanager are shared between threads and the tasks running in
those threads might compete over them. The choice of switching the used state backend
from the in-memory to the RocksDB state backend had a great positive impact on the
scalability of use cases that need to manage large state. We observed this with hierarchical
sensor aggregation (use case 2) where we scaled the workload by enlarging the sensor
configuration which has to be stored in operator state. For time attribute based aggregation
(use case 4) we discovered similar improvements as this use case needs to manage very
large time windows with accordingly large state. With a use cases that only handles smaller
state, such as downsampling (use case 3), we observed that the in-memory backend yields
better performance and therefore scalability because the RocksDB backend comes with a
performance overhead that is not worth taking for such small states.

6.2 Future Work

The execution of the benchmarks with all configurations and sub-benchmarks took a large
amount of time. By testing all combinations of 5 instance values and 6 workload values for 5
minutes each, results in an execution time of 2.5 hours per use case and configuration. This
time could increase even more for two reasons. First, we suspected that a sub-benchmark
duration of more than 5 minutes could lead to more accurate results. And second, for
benchmarking additional configurations and executing benchmarks at larger scale, the
number of sub-benchmarks would need to increase drastically. At some point the time and
cost of benchmarking might not be feasible anymore, which is why it is very important to
minimize the number of sub-benchmarks by using real time analysis while the benchmarks
are running and a heuristic to identify unnecessary benchmarks. For example, when a
workload could be processed by some number of instances, it does not make sense to test
if more instances can still handle that workload. Similarly, when a number of instances
could not handle a workload, it is not necessary to test if it can handle larger workloads.
Online analysis during benchmark execution and suitable heuristics to reduce the number
of experiments are currently being worked on.

With improved experiment execution time, benchmarks of more Flink configurations
and at higher scale with more instances and higher workloads can be carried out more
efficiently. Also, the scalability of the use cases regarding other workload dimensions
should be explored.

Scalability benchmarks of other stream processing engines such as Spark, Storm or
Samza, could be realized as well in the future. Moreover, future work should include the
comparison of scalability benchmarking results from different stream processing engines.


Bibliography


