Scalability Benchmarking of Kafka Streams
Deployment Options

Björn Vonheiden
Software Engineering Group, Kiel University, Germany
stu202077@mail.uni-kiel.de

Abstract. Kubernetes, Kafka, and Kafka Streams are commonly used together in software systems. The Theodolite method allows to benchmark the scalability of such systems. Kafka Streams provides many configuration options. Thus, it is difficult to set the configuration options in the best possible way. Previous research with Theodolite, however, considers only few configuration options for the scalability. Therefore, we explore further configuration options for Kafka and Kafka Streams in this paper. On one hand we apply the topology optimization option, on the other hand we focus on high availability and resiliency options. We define different parameters for these configuration options and execute the Theodolite benchmark with them. Ultimately, we present and discuss the results of the benchmark executions and identify the influence of the parameters regarding the scalability of Kafka Streams applications.

Keywords: Theodolite · Scalability Benchmark · Kafka Streams

1 Introduction

Many software systems are built around messaging systems like Apache Kafka. Microservices process data from these systems “by transforming, aggregating, and joining with other data” [6] and use them to communicate asynchronously. In order to process data from Kafka, stream processing engines like Kafka Streams or Apache Samza can be used. Engines need to process Big Data and, therefore, need to be scalable to handle these loads efficiently. To scale an application, multiple instances of its microservice can either process part of the data (data parallelism) or part of the operations (task parallelism). Scalability can be benchmarked and analyzed with Theodolite [6].

Kafka and Kafka Streams provide many configuration options for customization [6,8]. In this paper, we explore (1) the impact of applying automatic optimization to the Kafka Streams topology and (2) configuration options that address high availability and resiliency [2]. Therefore, we execute benchmarks for different configurations and analyze how the configurations influence the scalability of Kafka Streams applications.

In Section 2 we explain foundations to understand this paper. We describe our approach for executing our benchmarks in Section 3 and present our results.
in Section 4. In Section 5 we focus on threats to our results. Finally, we give an overview of the benchmark results and point out future work in Section 6.

2 Foundations

This section covers the necessary foundations to understand this paper. It includes the different systems and frameworks that are used.

2.1 The Distributed Event Streaming Platform Kafka

Kafka [1,7] is a distributed event streaming platform. It collects and delivers messages with high volume and low latency. It is designed to be scalable and to support a high throughput of data. The usage is similar to a messaging system. A Kafka cluster typically consists of multiple brokers. These are server processes that store messages. A topic is a stream of messages and can be split into multiple partitions that are distributed among the brokers. Each partition contains distinct data of its corresponding topic. There are also Producers that publish messages to topics and consumers that pull messages from topics. One or more consumers form a consumer group whose members jointly consume messages of subscribed topics. Multiple consumer groups can subscribe to the same topic and each of them consume the full set of messages. Inside a consumer group, every partition of a topic is assigned to one consumer and messages of this partition are only consumed by it.

2.2 The Kafka Streams API

The Kafka Streams API [1,9] is a stream processing engine that enables to develop applications, which process data streams. Immutable records that form an append-only sequence are called data stream [9]. The API can be used for creating an application, which can read from and write to topics and apply stateless and stateful operations to the data. Applications can have multiple instances, which process the data. The different instances share the same application ID and consumer group. This way, the partitions are split between the instances and each instance processes only part of the input topic data (data parallelism) [9,11].

2.3 The Theodolite Scalability Benchmarking Framework

Benchmarks are a measuring instrument in empirical software engineering [10]. In Theodolite they are used to determine the scalability of stream processing engines [6]. Scalability means “the ability of a system to continue processing an increasing load with additional resources provided” [4].

Theodolite defines four different use cases that are implemented as microservices using stream processing engines. These use cases are derived from the Titan Control Center [5], an analytics platform for Industrial Internet of Things (IIoT)
data. The use cases are implemented with multiple stream processing engines. Every use case reads data from an input topic from Kafka. Use case UC1 transforms the data into a different data format. In use case UC2 the frequency of the data is downsampled. Data that arrives in a defined time window is aggregated and a single result is produced for this window. Hence, the frequency of data is reduced. With use case UC3 messages are aggregate based on the same temporal attributes. This temporal attributes can be e.g. the hour of the day (e.g., 8h-9h) or the day of the week (e.g., Tuesday). Lastly, use case UC4 performs a hierarchical aggregation [3]. For this the use case gets a tree that describes a hierarchy in which leaves are the keys of the input data and nodes are groups for which the data of the children should be aggregated.

Ultimately, the Theodolite method determines how the number of instances increases when the load increases. An instance refers to a single processing instance of the microservice for a use case. The load can have different dimensions. In our case we vary the amount of different keys (UC1, UC2, and UC3) or the number of nested groups (UC4). More specific details about Theodolite are presented by Henning and Hasselbring [4,6].

3 Execution

3.1 Setup

The benchmarks are executed in a Kubernetes cluster (version 1.18) that consists of 5 nodes. Each of the nodes has 348 GB RAM and $2 \times 16$ CPU cores. Thus, there are in total 160 cores. The nodes are connected with 10 Gbit/s Ethernet.

10 Kafka brokers, 3 Zookeeper instances, and one Confluent Schema Registry instance are deployed to enable the Theodolite benchmark. Table 1 shows the default values for Kubernetes, Kafka, and Kafka Streams that are used for the benchmark executions. In order to make the results comparable with previous benchmarks, the same default values are used. The number of partitions defines the maximal parallelism of Kafka Streams applications [5]. Use cases are executed with a defined amount of instances. Each instance is deployed as a Kubernetes pod and gets the same amount of CPU cores and memory.

3.2 Benchmarks

Table 2 shows the Kafka Steams configuration options that are evaluated in this paper with Theodolite. The topology optimization allows Kafka Streams to apply optimizations to the topology. The replication factor, number of standby
replicas, and maximum warmup replicas are configuration options to enhance high availability and resiliency. Setting the replication factor to 3 is recommended by the documentation for resiliency. This way, Kafka Streams create internal topics with the replication factor 3. Internal topics are created and internally used by Kafka Streams. As recommended by the documentation, we set the same replication factor on source topics.

We execute the benchmark for each use case with different numbers of instances, different loads, and the defined values for evaluation. We use heuristics from Theodolite for the execution of the benchmarks and, therefore, reduce the number of executed experiments in a benchmark. Instead of executing every load with every number of instances, we apply linear search and domain restriction. The linear search corresponds to heuristic H1 and domain restriction to heuristic H3 from Henning and Hasselbring. For a given load we execute an experiment with an increasing number of instances. When we find the first N instances that can handle the load, we continue with the next higher load. We start for the next load with N instances and repeat the previous step.

Sufficient instances are provided for a load if the slope of the record lag is below a defined threshold. The record lag describes how many messages in a topic are not consumed at a point in time and the slope of the record lag indicates by how many messages per second the record lag increases or decreases. For UC1, UC2, and UC4 the threshold of the slope is 2000, which is also used by Henning and Hasselbring. The threshold for use case UC3, however, is set to 100, because of the smaller load.

Afterwards we perform the analysis for the executed benchmarks. We then compare the results of a configuration option with the default execution. The same threshold as above is used to identify if the load is handled by the instances.

4 Results and Discussion

This section shows and discusses the results from Section 3. The use cases are executed once or twice for each configuration value. As mentioned in Section 2.3 we have different load dimensions. The load for use cases UC1, UC2, and UC3 is defined with the number of data sources, which are the number of different keys. We generate one message per key and second. Therefore, the number of data sources corresponds to the throughput in messages per second. For UC4, it is different, we rely on the number of nested groups.

We provide a replication package, the measurements, and the results to replicate our work.
4.1 Topology Optimization

The topology optimization reduces the number of internal topics. Figure 1 shows the results for the executions with the topology optimization enabled.

For use cases UC1, UC2, and UC3 the results are almost similar for the execution with the default configuration and with the topology optimization.

Use cases UC2 and UC3 have punctual strong increases for the required number of instances. UC2 has an increased demand of instances for the step from 350k to 400k data sources and for the executions Default #2 and Optimization #2 from 450k to 500k data sources. The same applies to use case UC3 from 9k to 10k data sources where an increase from 10 instances to at least 19 instances is observed. For 6, 7, and 8 nested groups in use case UC4, both enabled and disabled topology optimization, require the same amount of instances. 16 instances are required with the topology optimization to handle the load of 9 nested groups. In contrast, the default configuration was not able to handle the load, even up to 120 instances.

The effects of topology optimization in the benchmarks depends on the load and the use case, i.e., which streams functions are used in the use cases. In use case UC1, UC2, and UC3 the topology optimization does not change the number of required instances to handle the loads and, therefore, also not the scalability ability of the use cases.
In UC4 the topology optimization has an impact on processing 9 nested groups. Even with 120 instances, the load is not handled with the default configuration. As described in Section 3, the trend slope needs a threshold below 2000 messages per second, to have sufficient resources for computation. 16 instances with topology optimization fulfill this for the load of 9 nested groups. However, while running the benchmarks we observed the number of produced messages is 10 times less than the expected number. Further, the message “Skipping record for expired window” occurs multiple times in the log. For the hierarchical aggregation windows of the length 5000 ms and grace period of 0 ms are defined. This means, records that fall in the same time window are aggregated and out-of-order records are discarded when the time window they belong to is already over [3]. Further investigation is needed to determine the reasons for this behavior and benchmarks may need to be repeated for this use case.

4.2 Topic Replication

The replication factor describes how often an internal topic should be replicated. A higher replication factor tolerates a higher number of broker failures, but in return increases the storage space and possibly the overhead to access the Kafka topics [1].

In Figure 2 the number of required instances for use cases UC1, UC2, and UC4 are nearly the same for the different executions. In UC2 less instances are required with the topic replication for 300k to 500k data sources, but it was not possible to handle 550k data sources with up to 40 instances and topic replication. For UC3 the executions are the same up to 3k data sources. From 4k data sources on the executions diverge and the topic replication needs more instances. A load higher than 9k data sources can not be handled with replication.

The processing itself is not changed for the Kafka Streams application, instead, the work for the Kafka brokers has increased. The replication happens per topic partition. Every topic partition has one leader and zero or more followers who are consumers of the leader and read data from the leader to their own logs. Producers write to and consumers read from the leader of the topic partition. The topic replication adds some overhead to the network and Kafka brokers and may influence the reads and writes for the Kafka Streams applications [1].

In our benchmarks the scalability is not influenced by the replication for use case UC1, UC2, and UC4. In UC4 internal topics exists, but the additional overhead on Kafka has a small impact. For use case UC2, the performance is for some loads even better with replication. Nevertheless, it can not handle the highest load in the benchmark. The performance increase could be due to caching. The followers read from the leader before the use case application and, therefore, the required data may be cached and the application gets a faster response. The additional network traffic and overhead for 550k data sources may be too large for the Kafka brokers and, thus, the time to pull the data increases.

Use case UC3 needs to maintain a state for every sliding window and key. Therefore, the state gets large and it needs to be written more to the internal
Scalability Benchmarking of Kafka Streams Deployment Options

Fig. 2: Kafka Streams scalability benchmark results with replication 3.

topics. This increases the load on Kafka and the scalability is not provided anymore for high loads. However, the problem may not be the scalability of the Kafka Streams application. Instead, the Kafka brokers might be overloaded and scaling up Kafka could resolve the problem.

4.3 Standby Replicas

Kafka Streams uses local state stores (embedded RocksDB instances) and Kafka topics to store the state from stateful operations. Standby replicas are copies of these local state stores on other instances. For every local state store, Kafka Streams tries to create the number of copies defined by the number of standby replicas. When using n standby replicas, n+1 Kafka Streams instances are required [1]. Hence, for one standby replica, at least two instances and for two standby replicas at least three instances are required.

Use case UC1 needs nearly the same amount of instances with or without standby replicas. Only small deviations of at most one instance appear in the benchmarks. For the use cases UC2, UC3, and UC4 more instances are required most of the time with standby replicas. UC2 needs many more instances (6, 15) with standby replicas for 450k data sources. In UC3, only the configuration with 1 standby replica is able to handle the load 14k. (Figure 3).

Use case UC1 performs a map on the input data and transforms it into another data format. This is a stateless transformation and, therefore, UC1 does
not maintain a local state store and has no standby replicas. The other use cases perform stateful operations and therefore have local state stores. Kafka Streams push state updates from the local state stores to so called changelog topics, which are internal topics. The copies of the state store read from the changelog topic to keep the local state store up to date. This adds additional load to Kafka and the Kafka Streams applications.

With standby replicas, more resources are required for the same load. Scalability is still provided, but the required number of instances increases faster for high loads in use case UC2.

4.4 Warmup Replicas

Warmup replicas are like standby replicas, but they are created on demand. When a task should be reassigned, Kafka Streams creates warmup tasks, which restore the state for the task. When the local state store has finished restoring, the task is migrated to the new instance [1].

For the use cases UC1, UC3, and UC4, only minimal fluctuation, of at most two instances, appear with different number of warmup replicas [12]. We can see the results of UC2 in the Figure 4. For the lower loads the number of instances are the same and for higher loads some fluctuations occur.

In our benchmarks, we have a stable number of instances and loads. Thus, the Kafka Streams instances shouldn’t need to rebalance. However, in UC2 some
fluctuations appear. We recognized that our monitoring showed sometimes less instances than expected. This especially happened for UC2 with more than 25 instances. In the logs we observed many rebalances for the tasks and also found instances that had no active task assigned. This can be the reason for the fluctuations.

In the end the resource demand of applications should not change and the scalability is maintained.

5 Threats to Validity

The benchmarks are executed twice for the default options and once or twice for the different configuration options. To increase the validity of the results, further repetitions of the benchmarks should be executed. Moreover, the benchmarks are executed only in our Kubernetes cluster. Therefore, our results cannot be used to infer the results for other Kubernetes clusters with different hardware configurations and, thus, should also be executed on them.

In Section 4.1, we mentioned that use case UC4 may not provide the correct service. This needs to be evaluated and executions should be repeated. Moreover, we mentioned in Section 4.4 that the number of instances are less than expected sometimes and multiple rebalances happen with more instances. This can influence the results, because it may be possible that some experiments were not executed with the right number of instances. The Kafka option `group.initial.rebalance.delay.ms`, which delays the initial task assignment, might allow to avoid this problem. In a test setting we set this value to 30 000 and executed UC2 with it. At first sight this fixed the problem and we also observed that less instances are required for the higher loads.

The use cases are executed for 5 or 8 minutes. This may not be enough time to get valid results. With more instances the time to start all instances and the time to assign the tasks take longer. Therefore, the initial lag on startup gets higher. For the analysis the slope is considered after the first minute and this should not influence the results. However, the time might not be long enough to deliver a stable result.

In the analysis a threshold of 2000 for UC1, UC2, and UC4 and 100 for UC3 is defined. This may not fit the requirements of handling the load.
6 Conclusions and Future Work

In this paper we explore how options for high availability and resiliency impact the scalability of Kafka Streams applications. Configurations of these options are mainly based on availability demands and not on performance requirements. Therefore, it may help to see how the performance is impacted and how many additional resources are required. Except for standby replicas in use case UC2 and topic replication in UC3, the additional resource requirement is low.

The topology optimization reduces the creation of internal topics and data duplication. In UC4 the optimization dramatically reduces the resources needed for high loads. Therefore it is recommended to use this setting by default, because there should not be any drawbacks.

In the future work, benchmarks could be executed with multiple configuration options together, e.g., combining all options for high availability and resiliency. In addition, Theodolite can be equipped with a new feature that automatically finds the best configuration options for handling a load for a given application.

References