

Empirical Scalability Evaluation of Window Aggregation Methods in Distributed Stream Processing

Master's Thesis

Björn Vonheiden

December 13, 2021

KIEL UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
SOFTWARE ENGINEERING GROUP

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

Eidesstattliche Erklärung

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

Kiel, 13. Dezember 2021

Abstract

Nowadays, data stream processing is a paradigm that is used to process large amounts of data in real time. Hopping window (also called sliding window) aggregations are a core operation in distributed stream processing.

In this thesis, we empirically evaluate the scalability of hopping window aggregations. Therefore, we benchmark different window aggregation methods. These are the native hopping window implementations of the stream processing engines Kafka Streams, Apache Flink, and Apache Spark. Further, we benchmark the window aggregation method from the Scotty framework that uses slicing. With sliding windows, Kafka Streams provides another window aggregation method that we evaluate. To empirically evaluate the scalability, we use the Theodolite benchmarking method. We apply two benchmark applications that implement the different windowed aggregation methods. One is an application benchmark from Theodolite and one is a microbenchmark from the Open Stream Processing Benchmark. In our evaluation, we execute benchmarks with different window configurations and evaluate the scalability of the window aggregation methods.

Our results show that all the methods are scalable. With the native hopping window implementations, Spark can process the highest loads, followed by Flink, and Kafka Streams can process the lowest loads. The number of overlapping windows influences the resource demand of the native hopping window implementations. Scotty can process higher loads than the native hopping window implementations of Kafka Streams and Flink. The sliding period of the hopping window influences the resource demand of Scotty. In the sliding window implementation of Kafka Streams, the rate of the processed data and the time difference determine the resource demand. If the number of overlapping windows is the same, the sliding window implementation of Kafka Streams can process higher loads than the native hopping window implementations of Kafka Streams and Flink.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Goals	2
1.2.1	G1: Theodolite Benchmark UC3 with new Implementations	3
1.2.2	G2: Microbenchmarks for Hopping Window Aggregation	4
1.2.3	G3: Execute Benchmarks	4
1.3	Document Structure	4
2	Foundations and Technologies	5
2.1	Distributed Stream Processing	5
2.2	Windowed Aggregation	5
2.2.1	Window Types	6
2.2.2	Windowed Aggregation Concepts	10
2.3	The Distributed Event Streaming Platform Apache Kafka	11
2.4	The Kafka Streams Stream Processing Framework	12
2.5	The Flink Stream Processing Framework/Engine	13
2.6	The Spark Analytics Engine	15
2.7	The Scotty Window Processor	17
2.8	Benchmarking Software Systems	18
2.9	The Theodolite Scalability Benchmarking Framework	18
2.9.1	The Theodolite Scalability Benchmarking Method	18
2.9.2	The Theodolite Scalability Benchmarking Operator	20
2.9.3	Benchmarks Provided by Theodolite	21
3	Application Benchmark	23
3.1	Selection of Benchmark	23
3.2	Flink with Scotty	24
3.3	Kafka Streams with Scotty	26
3.4	Kafka Streams with Sliding Window	29
3.5	Integration into Theodolite	30
4	Microbenchmark	31
4.1	Selection	31
4.1.1	Selection Criteria	32
4.1.2	Benchmark Selection	35
4.2	The Open Stream Processing Benchmark (OSPBench)	35

Contents

4.3	OSPBench Integration into Theodolite	36
4.3.1	Workload Generator Integration into Theodolite	36
4.3.2	Kafka Streams SUT Integration into Theodolite	39
4.3.3	Flink SUT Integration into Theodolite	43
4.3.4	Spark SUT Integration into Theodolite	44
4.3.5	Theodolite Integration Summary	47
5	Experimental Evaluation	49
5.1	Theodolite Extensions	49
5.2	Methodology	50
5.3	Results and Discussion	54
5.3.1	Comparison of the Frameworks	55
5.3.2	Kafka Streams Scalability	58
5.3.3	Flink Scalability	63
5.3.4	Spark Streaming Scalability	67
5.4	Threats to Validity	68
5.4.1	Internal Validity	68
5.4.2	External Validity	71
6	Related Work	73
6.1	Scalability Benchmarking of SPEs	73
6.2	Optimized Hopping Window Implementations	74
7	Conclusions and Future Work	77
7.1	Conclusions	77
7.2	Future Work	77
	Bibliography	79

List of Acronyms

<i>API</i>	application programming interface
<i>IIoT</i>	Industrial Internet of Things
<i>IoT</i>	Internet of Things
<i>OSPBench</i>	Open Stream Processing Benchmark
<i>SLO</i>	service-level objective
<i>SPE</i>	stream processing engine
<i>SUT</i>	system under test
<i>YSB</i>	Yahoo Streaming Benchmarks

Introduction

1.1 Motivation

Nowadays, data stream processing is a paradigm that is used to process large amounts of data in real time. Modern stream processing systems process continuous data streams by chaining software components (called operators) that, e.g. filter, map, or aggregate the data [Sax et al. 2018]. The aggregation over time windows is a core operation in distributed data stream processing [Traub et al. 2021]. Windows are bounded subsets of a continuous data stream and allow aggregations to be performed on them.

For example, LinkedIn, a global social network company, publishes more than trillions of events per day in their messaging system and processes this data [Noghabi et al. 2017]. In LinkedIn's processing applications it is common to perform stateful computations such as windowed aggregations. Noghabi et al. [2017] present the Email Digestion System (EDS) as an example for stateful computations. EDS is responsible for sending emails to users containing all updates of a certain period of time in a single email. Therefore, it aggregates all updates over time windows. Further, the EDS computes the effectiveness of digested emails which also use windowing.

Another example that uses windowed aggregations is the Titan Control Center [Henning and Hasselbring 2021c]. It is a platform for analyzing industrial energy consumption [Henning et al. 2021a]. Different functionalities are provided for the analysis of data streams from Industrial Internet of Things (IIoT) sensors. One function is the aggregation of data, which includes downsampling, aggregation of data points with the same temporal attribute, and hierarchical aggregation of sensors into groups. All of these aggregations utilize windowing.

Both of the preceding examples use hopping window aggregations. A hopping window (also called sliding window) is defined by a window size and a sliding period [Akidau et al. 2015]. Incoming data is split into windows based on the window size and then aggregated using a user-defined function. The sliding period defines how often a new window starts and, if it is smaller than the window size, multiple overlapping windows exist. A naive implementation can lead to a high degree of redundant computations [Carbone et al. 2018; Traub et al. 2021]. That is the case when data belongs to multiple overlapping windows and is aggregated in each of them. Scotty, an operator for hopping window aggregations,

1. Introduction

introduces general stream slicing that offers a solution to omit redundant computations [Traub et al. 2021].

Another promising method for aggregating time windows was recently presented as part of the Kafka Streams framework [Apache Software Foundation 2021c]. Kafka Streams introduced sliding windows, which provide similar semantics to hopping windows. The sliding window aggregates data based on a window size and does not need a sliding period. It continuously slides over the data and creates every possible unique combination of windows for the data. Based on the sliding period of the hopping window and the message frequency of the data, the sliding window produces less, equal, or more overlapping windows.

A stream processing engine (SPE) should handle large amounts of data and, therefore, uses distributed processing. The data is usually split based on keys so that multiple processing instances can process a part of the data. Hence, the system must scale horizontally [Noghabi et al. 2017]. Windowed aggregation is a core operation in SPEs and, thus, scalability is also an important quality for it. Many benchmarking studies exist to evaluate the performance of SPEs [Bordin et al. 2020; van Dongen and Van den Poel 2021b; Hesse et al. 2021]. The studies use different metrics like throughput, latency, or scalability and benchmark different aspects of SPEs. However, to the best of our knowledge, there is not any benchmark study that assesses the scalability of hopping window aggregations.

With this thesis, we empirically evaluate the scalability of incremental hopping window aggregations. We evaluate the scalability of Scotty and Kafka Streams sliding windows, and compare them to the native framework implementations for hopping window aggregations. Moreover, we select a microbenchmark that uses hopping window aggregations to evaluate its scalability.

We use benchmarks for empirical evaluation. They can be used to evaluate and compare software systems [Ralph et al. 2021]. Moreover, they provide repeatable, objective, and comparable results by defining standardized measurements [Hasselbring 2021]. We use Theodolite [Henning and Hasselbring 2021d] as the benchmarking tool and benchmark applications from Theodolite and the Open Stream Processing Benchmark (OSPBench) [van Dongen and Van den Poel 2020; 2021, a; b].

1.2 Goals

Our primary goal is to empirically evaluate the scalability of hopping window aggregations in distributed stream processing. Therefore, we use benchmarking, an empirical method in software engineering research [Ralph et al. 2021]. We use Theodolite as our benchmarking tool. Theodolite provides metrics for scalability, enables the replication of experiments, and allows the use and configuration of different load generators and systems under test (SUTs).

With goal G1, we provide new implementations for a task sample that uses hopping window aggregations. We want to find an existing microbenchmark that uses hopping

windows and integrate it into Theodolite with goal G2. Finally, with goal G3, we execute benchmarks on the new SUTs and evaluate the results to answer our primary goal.

1.2.1 G1: Theodolite Benchmark UC3 with new Implementations

Theodolite [Henning and Hasselbring 2021d] comes with a set of four benchmarks. One of these benchmarks is UC3, which performs a hopping window aggregation over time attributes. The specific time attribute in UC3 is the hour of day. By default, the window size is 30 days and the sliding period is 1 day. Thus, every day a new window starts, and 30 overlapping windows exist. Though, the window size and the sliding period can be configured. In the Kafka Streams and Flink implementations, arriving messages are aggregated in each of the belonging windows.

Goal G1.1 addresses the problem of redundant aggregations in Kafka Streams and Flink. With goal G1.2, we use another type of window for the aggregation.

G1.1: Scotty Window Processor

The default implementations for hopping window aggregations in Kafka Streams and Flink perform redundant computations for overlapping windows. In contrast, Scotty uses another technique, called stream slicing, for window aggregations [Traub et al. 2021]. Scotty splits the data stream into distinct slices based on the window size and sliding period. Then for each slice, partial aggregates are computed. When a window ends, the partial aggregates of the contained slices are combined and a final result is computed. Hence, redundant computations should be reduced. Therefore, we implement Scotty as another approach in the existing Kafka Streams and Flink implementations of UC3.

G1.2: Kafka Streams Sliding Window Aggregation

The task sample UC3 uses hopping windows for the aggregation. With the default configured sliding period, every day a new window starts. When a window is finished, it contains the aggregated data of the most recent 30 days. However, the next window contains only 29 days of data, and in between those two windows, one day of data is missing.

To get more up-to-date data, the sliding period can be reduced. Hence, window results are published more often. In the default setting, the load generator generates new data every second. Therefore, a sliding period of 1 second would deliver the most recent data.

Kafka Streams has another type of window, the sliding window, which provides the functionality to get the most recent data. A sliding window has a time difference and continuously slides over the time axis. For each unique combination of data, a window is created. We implement the sliding window with Kafka Streams to provide another approach to perform the windowed aggregation of task sample UC3.

1. Introduction

1.2.2 G2: Microbenchmarks for Hopping Window Aggregation

The existing task samples in Theodolite are derived from Titan, an IIoT platform [Henning and Hasselbring 2021c]. Many other benchmarks for stream processing exist that define their qualities, metrics, workloads, and task samples. However, most of them do not consider scalability. Thus, it would be interesting to benchmark one or more of them with Theodolite to evaluate the scalability. To aid our main goal we look particularly at hopping window aggregations. Further, we prefer a microbenchmark over an application benchmark. A microbenchmark allows a better estimation of the scalability of the hopping window aggregation. As a side benefit, the integration of the benchmark would assess how extendable Theodolite is.

1.2.3 G3: Execute Benchmarks

Goal G1 and goal G2 provide new SUTs for which no scalability evaluations are available. Therefore, we execute these SUTs with Theodolite. We define suitable loads and resources for the scalability evaluation and execute the benchmarks. Then, we discuss the results of the benchmark executions and use plots to answer our primary goal.

1.3 Document Structure

In Chapter 2, we provide the foundations and technologies required for this thesis. Chapter 3 shows the implementations required for goal G1, and we describe the selection and integration of a microbenchmark in Chapter 4. To achieve our main goal, we perform the evaluations in Chapter 5. Chapter 6 addresses related work on scalability benchmarking and windowed aggregations in distributed stream processing. Finally, in Chapter 7 we draw conclusions and provide an outlook for future work.

Foundations and Technologies

This chapter covers the foundations and the technologies we use in the rest of the thesis.

2.1 Distributed Stream Processing

In sensor networks, location-tracking services, and other Internet of Things (IoT) applications, multiple devices generate data and push them asynchronous to servers, which can lead to high volume data streams [Cherniack et al. 2003]. Stream processing applications should process these data streams in a timely and responsive fashion. Often these applications are distributed.

Cherniack et al. [2003] and Sax et al. [2018] define data streams similarly. In Cherniack et al. [2003] “a data stream is a potentially unbounded collection of tuples” generated in real time, and Sax et al. [2018] describe it as an “append-only sequence of immutable records.”

Stream processing applications utilize streaming operations to process the data streams. Streaming operations can be distinguished into stateless and stateful operators [Sax et al. 2018]. On the one side are **stateless operators**, such as filter, map, and flatMap. The function they define is applied to each record. On the other side are **stateful operators**, such as aggregation and join. They maintain a state and apply the operations to a record and a state. Therefore, they can consider more than one record for processing.

2.2 Windowed Aggregation

Sax et al. [2018] state that “aggregations in stream processing are usually based on a grouping attribute [...] and time windows”. In the following, we refer to windowed aggregation as a composition of windowing and aggregation.

With **windowing**, a dataset is sliced into chunks of finite duration for processing as a group [Akidau et al. 2015]. Akidau et al. [2015] distinguish between aligned and non-aligned windows. Aligned windows are applied to all the data. Whereas unaligned windows are only applied to specific subsets (e.g. per key) of the data.

The **aggregation** is a stateful operator (cf. Section 2.1). It is applied per grouping attribute (e.g. key) and window [Sax et al. 2018] and can be performed incrementally. Traub et al. [2021] classify aggregations into three types: distributive, algebraic, and holistic.

2. Foundations and Technologies

Table 2.1. Window names in SPEs [Apache Software Foundation 2021c; b; a].

Description	Kafka Streams	Apache Flink	Apache Beam
Consecutive windows with fixed size	Tumbling Time Window	Tumbling Window	Fixed Time Window
Fixed window size with sliding period	Hopping Time Window	Sliding Window	Sliding Time Window
Fixed window size, continuously slides	Sliding Time Window	<i>not available</i>	<i>not available</i>
Windows based on an inactivity gap	Session Window	Session Window	Session Window

Distributive aggregations have partial aggregates that are equal in type to the final aggregate and are fixed in size. The final aggregate gets computed by the partial aggregates. Examples are sum, min, and max. In **algebraic** aggregations, partial aggregates have a fixed size and summarize intermediate results. The intermediate results are used to compute the final aggregate. For example, average is such an aggregation. The sum and count are saved as the intermediate result and the final aggregate is computed by $\text{average} = \text{sum}/\text{count}$. In **holistic** aggregations, the size of partial aggregates is unbounded. For example, the median is a holistic aggregation. It requires storing all input values to calculate the median.

2.2.1 Window Types

Aggregations can be performed on different types of windows. Furthermore, these different window types have different names in different frameworks. Table 2.1 lists the names for Kafka Streams, Apache Flink, and Apache Beam. In the rest of the paper, we use the naming convention of Kafka Streams since the other frameworks do not have a name for the sliding time window of Kafka Streams. In the following, we describe the semantics of the windows. Thus, the descriptions abstract from concrete implementations.

Hopping time windows

Hopping time windows are time-based, have a fixed *window size*, and a *sliding period*. [Jafarpour and Desai 2019]. When the sliding period is smaller than the window size, the windows are overlapping [Akidau et al. 2015]. Figure 2.1a illustrates an example of a 5 minutes hopping window with a sliding period of 1 minute. The green and blue rectangles in the top are data records and form the data stream. Data records with the same color have the same record key. The time line denotes the stream time in minutes. Windows are depicted by the rounded rectangles and the border color indicates to which key the windows belong. The window size is 5 minutes and, therefore, each of the windows have a length of 5 minutes. A new window is added every minute, because of the sliding period. The sliding period is smaller than the window size and, thus, multiple overlapping windows exist. A data record is assigned to every window that exists at the time of the data record. The data record at time 0 is only added to the first window. In contrast, the green data record at time 5 is assigned to all 5 windows that exist at that time. Further, the

2.2. Windowed Aggregation

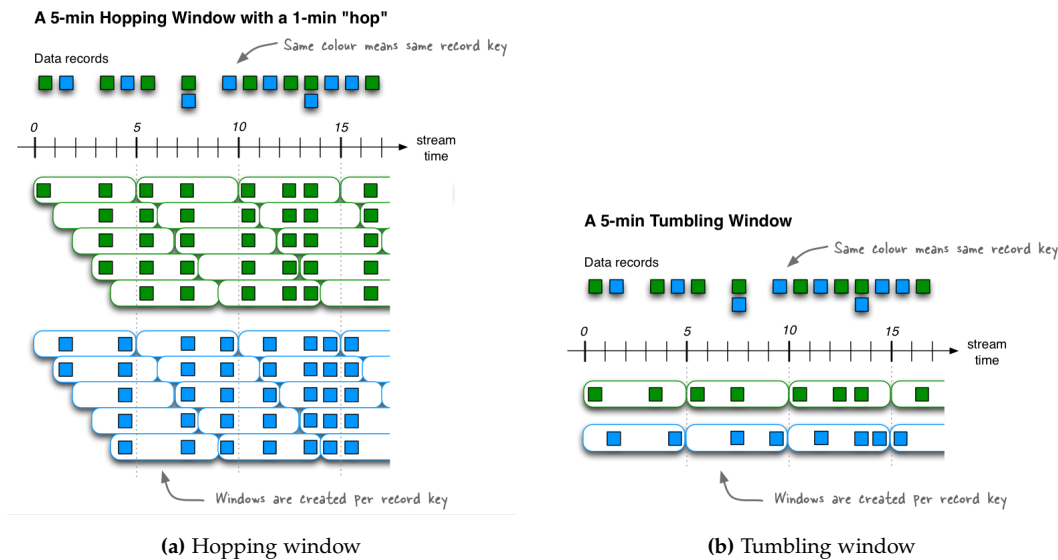


Figure 2.1. Hopping and tumbling window semantics in Kafka Streams [Apache Software Foundation 2021c].

windowing is grouped by the key. Hence, blue records are added to the blue windows and green records are added to the green windows.

Tumbling time windows

Tumbling time windows are a special case of hopping windows [Akidau et al. 2015] where the window size and the sliding period are the same. Thus, the windows do not overlap and when one window ends, the next one starts and there is no gap in between [Jafarpour and Desai 2019]. Figure 2.1b shows an example for a 5 minutes tumbling time window. A window has a length of 5 minutes and when one window ends the next one starts. Therefore, there are no overlapping windows and a record belongs to only one window.

Sliding time windows

Sliding time windows are fixed in size and defined by a *time difference* [Apache Software Foundation 2021c]. The window slides continuously over the time axis and two records are included in the same window, if their time difference is within the window. One record may fall into multiple snapshots of the sliding window and, therefore, there might be overlapping snapshots. However, each unique combination of records appears in only one sliding window snapshot.

2. Foundations and Technologies

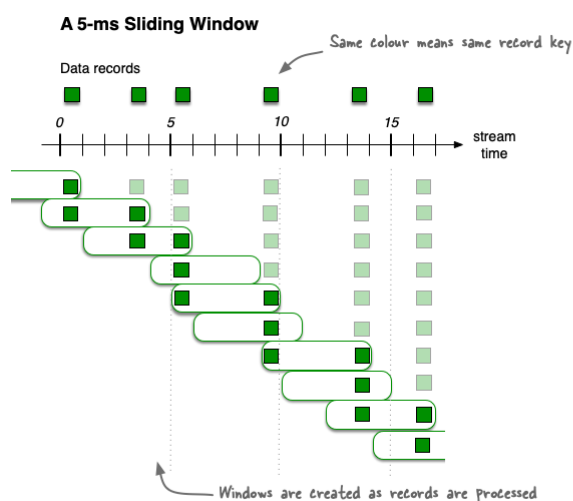


Figure 2.2. Sliding window semantics in Kafka Streams [Apache Software Foundation 2021c].

Figure 2.2 shows an example sliding window with a 5 ms time difference. The sliding window is fixed in size and, therefore, all rounded rectangles have a length of 5 ms. Further, no window contains the same data as other windows. The data record at time 3 is added to the end of the second window. At time 6 the data record at time 0 falls out of the window and the new record at time 6 is added. Then the window continues to slide over the axis and at time 8 the data record at time 3 falls out of the window.

In Figure 2.3a and Figure 2.3b, sliding and hopping windows are compared. Both examples use the same data stream. In Figure 2.3a, a time difference of 10 ms is used for the sliding window and, in Figure 2.3b, a window size of 10 ms with a sliding period of 1 ms is used for the hopping window. 1 ms is the smallest atomic time and, thus, the hopping window mimics the functionality of the sliding window [Gomes et al. 2021; Apache Software Foundation 2021e]. In the sliding window example, 7 windows are created which all have unique data combinations. 26 windows are produced with hopping windows and many windows contain redundant data.

Session windows

Session windows are session-based and non-overlapping [Jafarpour and Desai 2019]. They are defined by a *gap of inactivity/timeout gap* [Akidau et al. 2015; Apache Software Foundation 2021c]. Records are added to a session if they fall in the inactivity gap of an existing session. Otherwise, a new session is created. Figure 2.4 gives an example of a session window with a 5 minutes inactivity gap. The time line denotes the stream time in minutes. In Figure 2.4a,

2.2. Windowed Aggregation

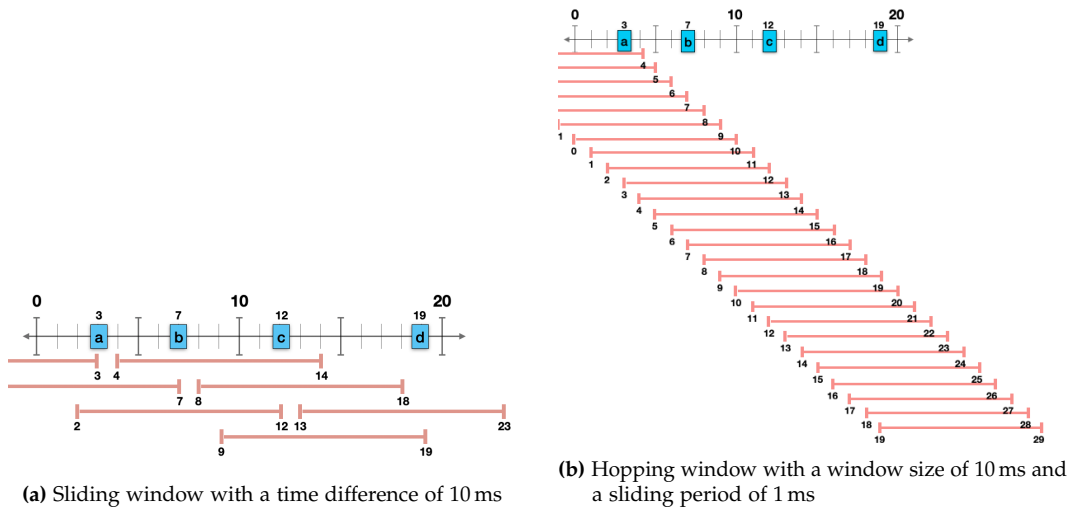


Figure 2.3. Comparison of sliding and hopping windows for processing the same data stream and providing the same functionality [Apache Software Foundation 2021e].

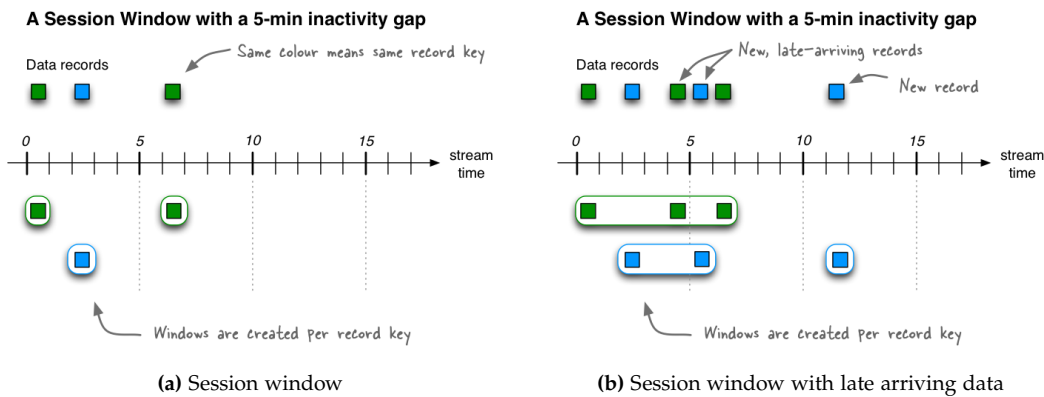

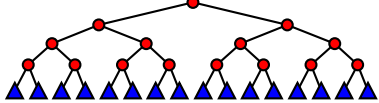
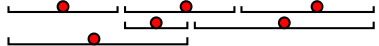




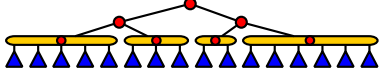


Figure 2.4. Session window semantics in Kafka Streams [Apache Software Foundation 2021c].

three data records arrive. The inactivity gap between the green records is greater than 5 minutes and, thus, a session is created for each record. In this case, session windows are created per key and, therefore, a new session is also created for the blue record. In Figure 2.4b, three new data records arrive with two of them being late data. The late green data record arrives at time 4, hence, it lies in the gap of inactivity of the session of the first green data record at time 0 and gets merged into it. In addition, the green record at time 6 is now also in the inactivity gap of the new session and is merged into the session. The late-arriving blue data record lies in the timeout gap of the existing session and, thus, is

2. Foundations and Technologies

	Memory Usage	Example
1. Tuple Buffer	$ \Delta \cdot \text{size}(\Delta)$	
2. Aggregate Tree	$ \Delta \cdot \text{size}(\Delta) + (\Delta - 1) \cdot \text{size}(\bullet)$	
3. Aggregate Buckets	$ \text{win} \cdot \text{size}(\bullet) + \text{win} \cdot \text{size}(\sqcup)$	
4. Tuple Buckets	$ \text{win} \cdot [\text{avg}(\Delta \text{ per win.}) \cdot \text{size}(\Delta) + \text{size}(\sqcup)]$	
5. Lazy Slicing	$ \bullet \cdot \text{size}(\bullet)$	
6. Eager Slicing	$ \bullet \cdot \text{size}(\bullet) + (\bullet - 1) \cdot \text{size}(\bullet)$	
7. Lazy Slicing on tuples	$ \Delta \cdot \text{size}(\Delta) + \bullet \cdot \text{size}(\bullet)$	
8. Eager Slicing on tuples	$ \Delta \cdot \text{size}(\Delta) + \bullet \cdot \text{size}(\bullet) + (\bullet - 1) \cdot \text{size}(\bullet)$	

Legend: ▲ Tuple ● Aggregate ● Slice with Aggregate \sqcup Bucket

Figure 2.5. Windowed aggregation concepts [Traub et al. 2021].

added to it. The new blue data record does not lie in the gap of inactivity of any session and a new session is created.

Hopping and tumbling time windows are aligned, i.e., all windows apply to all data in the defined period. Sliding and session windows are unaligned [Akidau et al. 2015].

2.2.2 Windowed Aggregation Concepts

Figure 2.5 shows concepts for windowed aggregations. The blue triangles are the tuples and the red dots are aggregates. We describe tuple buffers and buckets. Slicing is described in Section 2.7.

Tuple buffers save the all incoming tuples for the windows [Traub et al. 2021]. The aggregation is performed lazily. This means that the aggregation does not start until a window ends. In case of overlapping windows redundant aggregations are performed.

2.3. The Distributed Event Streaming Platform Apache Kafka

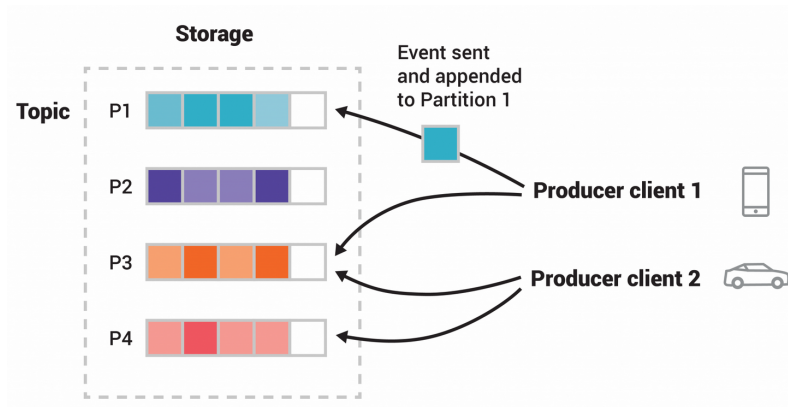


Figure 2.6. A Kafka topic with four partitions and two producers [Apache Software Foundation 2021c].

With **buckets**, a bucket is defined for each window. Incoming tuples are assigned to one or more buckets (i.e. windows) based on their time. Buckets are distinguished between *tuple buckets* and *aggregate buckets*. Tuple buckets save the tuples in the buckets. Thus, a tuple might be saved multiple times if the buckets overlap. The aggregate buckets incrementally aggregate the tuples and store partial aggregates. Redundant aggregations are performed for overlapping buckets.

2.3 The Distributed Event Streaming Platform Apache Kafka

Apache Kafka is a distributed messaging system [Kreps 2011; Jafarpour and Desai 2019]. It provides a publish/subscribe mechanism and is able to handle high volumes of data with low latency. To achieve this, it is distributed, scalable, and offers high throughput.

Kafka messages, also called records or events, include a key and a value [Sax et al. 2018; Jafarpour and Desai 2019]. Further, the messages have an embedded timestamp and an offset [Sax et al. 2018]. A topic is used to store messages of a particular type [Kreps 2011]. Topics are further divided into partitions to balance the load. Figure 2.6 illustrates how a topic is partitioned and the messages distributed. The shown topic has four partitions. Events are pictured as small squares and their keys are denoted by the color of the message. Every event is assigned and appended to a partition based on its key and events with the same key are in the same partition. Further, a partition is ordered and immutable [Jafarpour and Desai 2019]. In Section 2.1, we define the term *data stream* and a topic can be seen as an implementation of it.

Kafka is a distributed system and typically consists of multiple brokers [Kreps 2011]. These brokers are server processes that store one or more topic partitions and, thus, the

2. Foundations and Technologies

messages [Kreps 2011; Jafarpour and Desai 2019]. Partitions can be replicated across the brokers to achieve fault tolerance [Jafarpour and Desai 2019]. For each partition, one broker is the leader and the other zero or more brokers with that partition are the followers.

Producers publish messages to Kafka topics [Kreps 2011; Sax et al. 2018; Jafarpour and Desai 2019]. The producer selects with the message key to which partition it sends the message. Further, all producers send messages with the same key to the same partition. As shown in Figure 2.6, multiple producers can write to the same topics and partitions.

Data is read from Kafka by consumers. A consumer subscribes to one or more topics and pulls the data from the brokers [Kreps 2011; Sax et al. 2018]. To consume data with multiple consumers together, Kafka has the concept of consumer groups [Kreps 2011; Jafarpour and Desai 2019]. The consumers in the group subscribe to a set of topics and each published message is consumed by only one consumer within the group. Consumers of one group can be in different processes or on different machines [Kreps 2011]. Further, consumers can be added or removed from consumer groups [Jafarpour and Desai 2019]. The Kafka group management protocol handles these cases and performs a rebalancing to assign the partitions to the newly formed group. Multiple consumer groups can exist and are independent of each other [Kreps 2011]. That means each group consumes all messages from the subscribed topics.

2.4 The Kafka Streams Stream Processing Framework

Kafka Streams is a Java library for scalable stream processing [Sax et al. 2018; Jafarpour and Desai 2019; Wang et al. 2021]. The library enables developers to create real time processing applications [Sax et al. 2018; Wang et al. 2021] that are highly scalable, elastic, distributed, and fault-tolerant [Jafarpour and Desai 2019]. Kafka Streams is built on top of Kafka. The data streams are stored in Kafka, and the same data model with key, value, timestamp, and offset is used for the messages. A Kafka Streams application defines a processing logic to process the data stream [Jafarpour and Desai 2019; Wang et al. 2021]. Processing operations are, e.g, transform, join, aggregate, and windowing.

A high-level DSL is contained in Kafka Streams to specify a processing logic [Sax et al. 2018; Wang et al. 2021]. It provides functions to read data from Kafka topics, transform streams into new streams, and write data back to Kafka topics [Sax et al. 2018]. Data streams are abstracted in the DSL by the interfaces `KStream` and `KTable`. A `KStream` is an abstraction of the data stream, whereas a `KTable` sees the data stream as a changelog stream. This means every message is seen as an update to a table.

The operations defined with the DSL are transformed into a processing topology [Jafarpour and Desai 2019; Wang et al. 2021]. The topology includes source, stream processor, and sink nodes that are connected. Further, the topology is divided into sub-topologies that consist of consecutive nodes, where no data shuffling is required [Wang et al. 2021]. A sub-topology uses the continuous processing model [Zhang et al. 2021] where one message at a time is processed [Jafarpour and Desai 2019]. The message is passed through all

2.5. The Flink Stream Processing Framework/Engine

the nodes in the sub-topology before the next record is processed. Between successive sub-topologies a reshuffling of data is required. Therefore, the upstream sub-topology writes to a repartition topic and the downstream sub-topology reads from it [Wang et al. 2021].

The Kafka Streams DSL provides stateful stream processing operations [Jafarpour and Desai 2019; Wang et al. 2021]. States of the processing are stored at the processing instances in so-called state stores. Furthermore, the state store is also replicated to Kafka as a changelog topic [Wang et al. 2021]. The changelog topic and the repartition topic for shuffling are internal topics and are abstracted away from the user.

For windowed aggregations, Kafka Streams provides hopping, tumbling, sliding, and session windows [Apache Software Foundation 2021c]. The hopping, tumbling, and sliding window implementations of Kafka streams are based on buckets. Further, hopping and tumbling windows are aligned to the epoch, i.e. the first window starts at timestamp zero and all the window boundaries are predetermined. In contrast, sliding windows are data driven and aligned to the data record timestamps. This means the window boundaries are not predetermined and are based on the data. The computations on the windows are continuously updated and the latest aggregation results are stored in a state store. However, intermediate results can be suppressed and, thus, only the final window result is emitted.

For the progression of time in Kafka Streams, the stream time is used [Apache Software Foundation 2021c]. The stream time is the maximum timestamp of the processed records and, thus, the time only advances when new data is processed. A timestamp is assigned with the `TimestampExtractor` interface to a data record. Based on the used `TimestampExtractor`, different time notions are applied, for example, event time, processing time, or ingestion time. A stream task can process one or more partitions of a topic and the stream time is advanced separately for every task. Furthermore, the stream task selects the record with the lowest timestamp from the assigned partitions for processing.

The Kafka stream application does not require any cluster infrastructure other than Kafka to run [Jafarpour and Desai 2019]. Multiple instances of the Kafka Streams application can run independently. The application can scale out by starting more instances. To distribute the load between the instances, the Kafka group management protocol is used.

2.5 The Flink Stream Processing Framework/Engine

Apache Flink is a system for processing streaming and batch data and is available as open source [Carbone et al. 2015]. The central paradigm of Flink is data stream processing. It is used for real time analysis, continuous streams, and batch processing. Flink works in combination with durable message queues like Kafka or Amazon Kinesis. Further, Flink provides different application programming interfaces (APIs) that provide different levels of abstractions [Apache Software Foundation 2021b]. These APIs are the *DataStream API*, the *Table API*, and the *SQL API*.

2. Foundations and Technologies

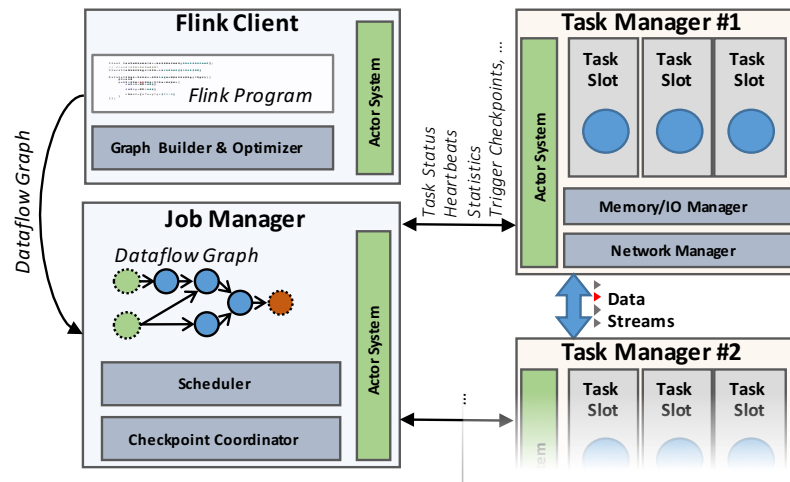


Figure 2.7. Flink cluster architecture [Carbone et al. 2015].

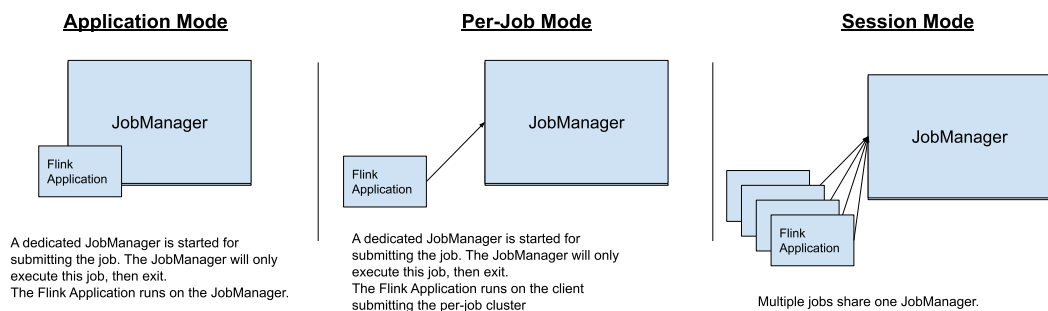


Figure 2.8. Flink deployment modes [Apache Software Foundation 2021b].

A Flink cluster consists of the Flink client, the job manager, and at least one task manager (Figure 2.7) [Carbone et al. 2015]. Program code is given to the client and compiled to a dataflow graph. This dataflow graph is submitted to the job manager that coordinates the distributed execution of the dataflow graph. Furthermore, the job manager coordinates checkpoints and recovery and monitors the state and progress of tasks. Tasks are executed on task managers in task slots.

The cluster can be deployed standalone by directly running the components on machines, in Docker, or in Kubernetes [Apache Software Foundation 2021b]. With a standalone cluster, the user needs to take care to start and restart the resources. Another approach is to use Kubernetes or YARN as resource providers. Flink is deployed directly to these resource providers and they handle the cluster creation.

2.6. The Spark Analytics Engine

Figure 2.8 shows different modes to deploy a Flink application. The different modes are application mode, per-job mode, and session mode. In the application mode, the Flink client runs on the job manager and the cluster is solely for the application. In the job mode, a cluster per job is started. This mode is only supported by YARN. In the session mode, multiple applications are managed by one job manager and share the same cluster.

The dataflow graph created from a program is a directed acyclic graph (DAG) [Carbone et al. 2015]. Nodes of the DAG are stateful operators and edges are streams that connect these nodes. The operators of the data flow graph are parallelized into subtasks, which enables parallel execution. Streams are also partitioned and they provide the data exchange between producing and consuming operators.

Flink utilizes stateful operators for processing [Carbone et al. 2015]. Stateless operators are a special case of stateful operators. The state of the operators is written to a durable storage. Event time, ingestion time, or processing time can be used for processing the data. For the progression of time, low watermarks are used [Carbone et al. 2015]. Watermarks are induced at the source of a topology and contain a time attribute t . They indicate the downstream operators that all events with a timestamp lower than t already entered the operators.

Windowing can be performed in Flink on keyed and non-keyed streams [Apache Software Foundation 2021b]. Non-keyed streams are processed by a single task. Keyed streams are split into logical streams based on a key selected from the data and, hence, can be processed by multiple tasks. A keyed stream requires a network shuffle and data is sent over the network to the downstream task. The windowing in Flink is implemented with buckets.

Like Kafka Streams, Flink uses a continuous processing model [Zhang et al. 2021]. Contrary to Kafka Streams it does not require Kafka to run. However, a job manager and one or more task managers are required to start a Flink application. Furthermore, systems other than Kafka can be used as sources and sinks for the data. Instead of using repartition topics, as is the case with Kafka Stream, Flink shuffles data across the network and sends it directly to downstream tasks. Kafka Streams and Flink use different approaches for the progression of time. Stream time is used in Kafka Streams and watermarks are used in Flink.

2.6 The Spark Analytics Engine

Apache Spark is a unified analytics engine and programming model for processing big data [Li et al. 2015; Apache Software Foundation 2021d]. An ecosystem exists around Spark and Spark supports machine learning, graph computation, SQL query, and streaming applications. The base concept of Spark is built around Resilient Distributed Datasets (RDDs) which are an abstraction of distributed memory [Zaharia et al. 2012a]. RDDs are collections of elements partitioned across a set of machines.

2. Foundations and Technologies

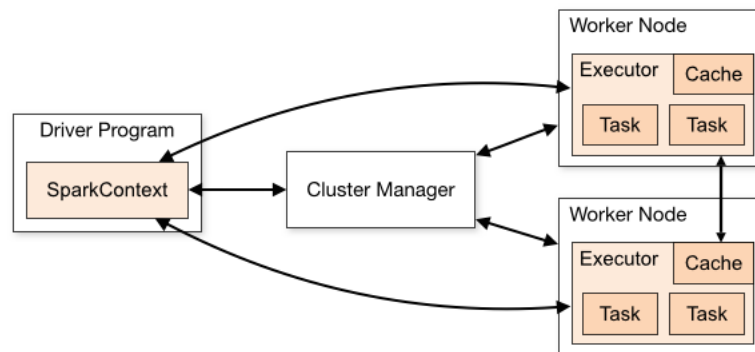


Figure 2.9. Spark cluster architecture [Apache Software Foundation 2021d].

Figure 2.9 shows the cluster architecture of Spark. It consists of a driver program, cluster manager, and worker nodes [Zaharia et al. 2012a; Apache Software Foundation 2021d]. The cluster manager allocates resources for applications. Developers write the driver program that defines one or more RDDs and the actions on them. The driver program requests the master for resources. It connects to the assigned workers in the cluster and acquires executors. The driver program sends the application code to the executors and, then, it sends the tasks to the executors to run them.

Spark offers two APIs for stream processing. One is Spark Streaming [Zaharia et al. 2012b] and the other is the newer Structured Streaming [Armbrust et al. 2018].

Spark Streaming uses a series of deterministic batch computations to perform stream processing [Zaharia et al. 2012b]. Therefore, the concept of discretized streams (DStreams) is used. In a D-Stream a series of RDDs are grouped together. Input data is received in small batch intervals and stored across the cluster. When a batch is ready, the data is processed in parallel. Stateless and stateful operators are provided for processing the data. These operators are applied to one or more parent DStreams and produce a new DStream. Stateful operators may process data of multiple time intervals and, therefore, may produce some intermediate RDDs as state. Furthermore, output operators are used to write data to external systems. Windowed operations combine the source RDDs that fall in their window size and produce the RDDs of the windowed DStream [Apache Software Foundation 2021d].

Instead of the continuous processing model used by Kafka Streams and Flink, Spark Streaming uses the batch processing model [Zaharia et al. 2012b]. Like Flink, Spark Streaming requires some components to create its own cluster. Moreover, Spark Streaming uses the processing time for the progress of time, i.e., the time the data gets ingested is used.

Structured Streaming provides a high level API that enables developers to write queries with the Spark *SQL API* [Armbrust et al. 2018]. Multiple streams and tables can be used in these queries and connectors provide a variety of input sources and output sinks. Structured

2.7. The Scotty Window Processor

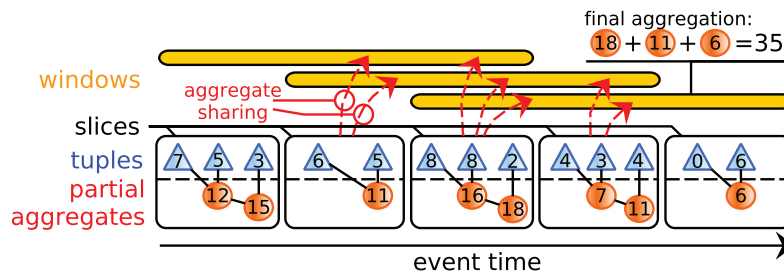


Figure 2.10. Example hopping window aggregation with stream slicing [Traub et al. 2021].

Streaming supports two execution modes. The default is the microbatch execution mode that uses the discretized streams execution model from Spark Streaming and the other is the continuous processing mode.

2.7 The Scotty Window Processor

Scotty [Traub et al. 2018; 2021] is a framework for window aggregations in stream processing. The framework provides tumbling, hopping, and session window aggregations. Scotty is available as an open source project¹ and provides connectors for different stream processing systems, for example, Kafka Streams, Flink, and Beam. Unlike other existing slicing-based techniques, Scotty provides complex window types like session windows, out-of-order processing, and the definition of user-defined aggregation functions [Traub et al. 2021].

Scotty utilizes the concept of stream slicing for the computation of windows and, therefore, should reduce redundant computations of overlapping windows [Traub et al. 2021]. To do this, the streams are split into non-overlapping slices and for each slice a partial aggregate is computed. Thus, redundant computations are avoided. When a window ends, the result is computed using the partial aggregates from the slices. It can be distinguished between lazy and eager stream slicing. Lazy stream slicing stores partial aggregates in the slices and slices are combined on demand. In the eager version, a tree additionally stores combinations of slices.

Figure 2.10 shows an example of using slicing for hopping window aggregations. The tuples (blue triangles) are the data and they form the stream. Slices are created based on the yellow windows. In each slice, the partial aggregates are computed. The partial aggregation is incrementally and is illustrated by a tree. The slices contained by a window, are aggregated to a final aggregation value. The red dashed arrow marks show where the partial aggregates are shared.

¹<https://github.com/TU-Berlin-DIMA/scotty-window-processor>

2.8 Benchmarking Software Systems

In software engineering research, benchmarks are used to compare specific characteristics of computer systems, databases, etc. [Sim et al. 2003; v. Kistowski et al. 2015]. Sim et al. [2003] define three components for benchmarks: the motivating comparison, the task sample, and the performance measures. The **motivating comparison** consists of the two concepts motivation and comparison. The motivation captures the need for the research area and the benchmark itself. Benchmarks are used for comparison, and, therefore, it must be clearly defined what is to be compared. A **task example** is used to test what the benchmarking tool or technique is designed to solve. Tasks should be representative samples of the problem domain. The **performance measures** are the measurements for the tests.

V. Kistowski et al. [2015] distinguish between three different categories of benchmarks: specification-based, kit-based, and hybrid. **Specification-based** benchmarks do not provide any implementations but describe the functions that must be achieved, the required input parameters, and the expected outcomes. In contrast, **kit-based** benchmarks provide implementations. A **hybrid** benchmark is the combination of the two. The desired key characteristics of benchmarks are relevance, reproducibility, fairness, verifiability, and usability.

Based on the size of the task sample, a distinction can be made between application benchmarks and microbenchmarks. In an application benchmark, the task sample models a complete application. With microbenchmarking, we mean the measure of performance for small code fragments, a single component or task [Laaber and Leitner 2018; Poggi 2019].

2.9 The Theodolite Scalability Benchmarking Framework

Theodolite [Henning and Hasselbring 2021d] is a method accompanied by an implementation for benchmarking the scalability of microservices and their employed stream processing frameworks. The ability of a system to continue processing an increasing load with additional resources is called scalability. Therefore, Theodolite defines two metrics for measuring the scalability [Henning and Hasselbring 2021b]. Theodolite provides different benchmark applications for scalability evaluations [Henning and Hasselbring 2020]. These applications are derived from the Titan Control Center, an analytics platform for IIoT data [Henning and Hasselbring 2021c].

Section 2.9.1 describes the basic concepts the Theodolite method is built on. Then, Section 2.9.2 shows how these concepts are implemented. Finally, Section 2.9.3 explains the defined SUTs of Theodolite.

2.9.1 The Theodolite Scalability Benchmarking Method

The Theodolite method evaluates how the computational resource demand evolves with an increasing load [Henning and Hasselbring 2020]. To show this, the scalability is expressed

2.9. The Theodolite Scalability Benchmarking Framework

by a function and plots are created that can show linear, quadratical or other scalability [Henning and Hasselbring 2021d]. Benchmarks in Theodolite are specification-based. That means, they are based on functional or business requirements.

The attributes load intensity, provisioned resources, and service-level objectives (SLOs) can be used to describe scalability [Henning and Hasselbring 2021b]. The **load intensity** describes the load induced on the SPEs. Usually, messages from a central messaging system are the load for SPEs. Theodolite uses Kafka as messaging system [Henning and Hasselbring 2021d]. The parallelized processing of data in SPEs is based on different keys for messages. Thus, the number of distinct keys is a sensible load dimension. In addition to the number of distinct keys, Henning and Hasselbring [2021d] outline various load dimensions. These are message frequency, time window size, number of overlapping windows, number of time attribute values, maximal number of elements, and maximal depth of nested groups. The number of different keys and the message frequency are configured in the workload generator. The frequency indicates how many messages are sent per key and time and the number of different keys how many distinct keys exist. Windowed aggregation loads can be configured by changing the window size and advance period. If the windowed aggregation is based on time attributes, various time attributes such as hour of day or the day of week can be configured.

Provisioned resources provides the set of resources that can be provisioned for processing the load [Henning and Hasselbring 2021b]. The SPEs utilize these resources to scale. SPEs are often deployed in containers and are scaled with the amount of containers (instances). Thus, the number of instances is one possible resource dimension.

SLOs define measurable quality criteria that an SPE should fulfill when processing the loads [Weber et al. 2014; Henning and Hasselbring 2021b]. Service level indicators (SLIs) define quantitative measures of some aspects of the level of service [Beyer et al. 2016]. An SLO defines “a target value or range of values for a service level that is measured by an SLI” [Beyer et al. 2016]. For example, the target of an SLO is defined as: $SLI \leq \text{target value}$, or lower bound $\leq SLI \leq \text{upper bound}$. In Theodolite, SLOs are functions [Henning and Hasselbring 2021b]. An SLO function returns a boolean value whether the SPE not violate the SLO with the induced load and the given resources. Theodolite provides one SLO that uses the lag trend metric as SLI. The **lag trend metric** indicates whether messages are queuing up at the message system. The lag indicates how many messages the consumers are behind the messaging system. Therefore, the lag is monitored during the execution of the SLO experiments. Afterwards, a trend line for the lag is computed using linear regression. The result is the lag trend that describes the average increase or decrease of the lag per second. Finally, the SLO checks whether the lag trend does not exceed a certain defined threshold.

Two metrics based on these attributes are defined: (1) resource demand metric and (2) load capacity metric [Henning and Hasselbring 2021b]. The **resource demand metric** is a function that maps load intensities to the minimum required instances that fulfill the specified SLOs. For example, the metric shows the increase in the resource demand when processing higher loads. The **load capacity metric** is a function that maps the provisioned

2. Foundations and Technologies

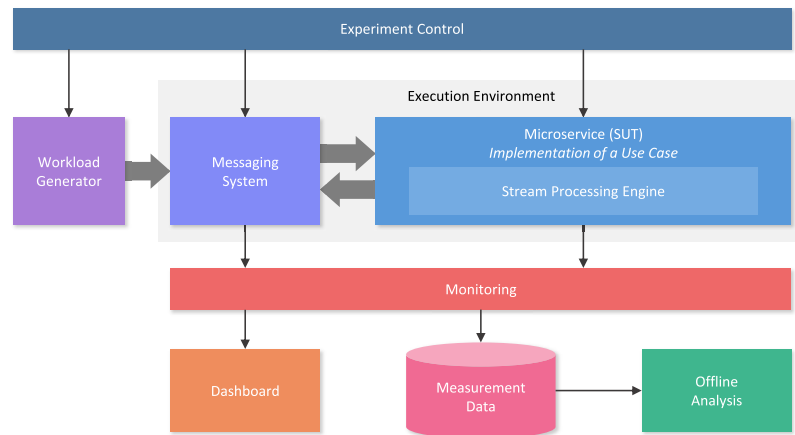


Figure 2.11. Theodolite framework architecture [Henning and Hasselbring 2021d].

resources to the maximum load intensity they can process. It shows at which rates the provisioned resources can process the data.

2.9.2 The Theodolite Scalability Benchmarking Operator

Figure 2.11 shows the architecture of Theodolite [Henning and Hasselbring 2021d]. Theodolite gets deployed in Kubernetes, a tool for declarative orchestration. The experiment control is responsible for running a benchmark. It is using the Kubernetes Operator Pattern [Henning et al. 2021b]. With the Operator Pattern, domain knowledge can be integrated into Kubernetes [Cloud Native Computing Foundation 2021]. The Operator Pattern requires custom resource definitions (CRDs) that define new custom resources (CRs) that are managed by the Kubernetes API. In addition, the operator runs as a controller in Kubernetes. The operator performs actions described by CR objects. Theodolite defines two CRs: (1) benchmark and (2) execution [Henning et al. 2021b]. A **benchmark** describes the SUT and the load generator. It specifies which Kubernetes resources are required for the deployment. Furthermore, possible load and resource types that can be used in the execution are defined. They are defined by a name and a list of patchers. Patchers are functions that take an input value and modify a Kubernetes resources according to the value. An **execution** is a single execution of a benchmark with one specific configuration. It uses one of the defined load and resource types from a benchmark and provide for each a set of values. One or more SLOs must be specified that define the properties that the SUT should satisfy. Further, the experimental setup, such as the execution time, warmup time, and repetitions, is configured.

The operator observe the Kubernetes API for new *execution* CR objects and creates SLO experiments for them. An **SLO experiment** determines for an SUT deployed with a

2.9. The Theodolite Scalability Benchmarking Framework



Figure 2.12. UC3 dataflow architecture [Henning and Hasselbring 2021d].

given number of resources whether a defined load can be processed by checking all SLOs [Henning and Hasselbring 2020]. Theodolite contains four search strategies that create these SLO experiments for an execution [Henning and Hasselbring 2020; 2021, a]. Moreover, a number of repetitions defines how often each SLO experiment is repeated. After the SLO experiments are executed for the defined number of repetitions, the median record lag of the experiments is chosen and checked against the defined SLOs.

Prometheus is used for monitoring the messaging system and SUTs [Henning and Hasselbring 2021b]. The monitored metrics collected by Prometheus are persisted and available for the offline analysis scripts. In addition, the metrics can be used in Grafana to observe the benchmarks in real time. Theodolite provides four SUTs implemented with different frameworks and provides corresponding workload generators [Henning and Hasselbring 2021d]. Moreover, users can provide arbitrary workload generators and SUTs as long as they can be deployed as containers and interact with Kafka.

2.9.3 Benchmarks Provided by Theodolite

In this section, we shall describe the predefined Theodolite task samples UC1, UC2, and UC4. As we want to modify UC3, as stated in Section 1.2.1, we describe it in more detail. The task samples are adopted from the Titan Control Center [Henning et al. 2021a] and described by a dataflow architecture. All the task samples interact with Kafka and are implemented with Kafka Streams and Flink [Henning and Hasselbring 2021d].

Task sample UC1 simulates database storage [Henning and Hasselbring 2021d]. Incoming messages are converted into another data format and written to a database. In task sample UC2, the incoming data is downsampled to a lower rate in order to reduce the number of events. Multiple records within a defined window are aggregated to one result. Tumbling windows (cf. Section 2.2.1) are used for this aggregation. In task sample UC4, messages are hierarchically aggregated. This means groups of sensors are aggregated together and multiple group levels may exist.

Task sample UC3 aggregates messages based on time attributes [Henning and Hasselbring 2021d]. A time attribute can be, for example, the day of week or day in the year. This functionality is implemented in Titan to model and identify seasonality in the power consumption data [Henning et al. 2021a]. For example, the average course of the consumption of the day can be computed with the hour of day time attribute.

The dataflow architecture of UC3 is shown in Figure 2.12. Data is read from an input stream and a new key is selected [Henning and Hasselbring 2021d]. The new key consists of the old key and a time attribute extracted from the timestamp of the message. These time

2. Foundations and Technologies

attributes are, for example, the hour of day or the day of week. Then, these new records are aggregated with hopping windows (cf. Section 2.2.1). The default window size is 30 days with an advance of 1 day. However, this can be defined arbitrarily.

Application Benchmark

In this chapter, we accomplish goal G1 and implement Scotty and the sliding window in Theodolite’s UC3. In Section 3.1, we justify the selection of the UC3 task sample and why we implement it with Scotty. The Scotty framework provides connectors for Apache Flink, Apache Storm, Apache Beam, Apache Kafka Streams, and Apache Spark. We go into the details of the implementation of Scotty into Flink and Kafka Streams in Section 3.2 and Section 3.3. Then, we describe the implementation of sliding windows into Kafka Streams in Section 3.4 and, finally, in Section 3.5, we show how we integrate these implementations into Theodolite.

3.1 Selection of Benchmark

For the selection of the benchmark, we go into more details about the relevance characteristics of benchmarks mentioned in Section 2.8. The relevance of a benchmark states how the results should be used and what relevant information it provides [v. Kistowski et al. 2015].

Windowed aggregation is a fundamental operation in stream processing [Traub et al. 2021]. It is used to perform aggregations on a continuous and unbounded stream. Kafka Streams and Flink use buckets for their hopping window aggregations. In contrast, Scotty uses the concept of stream slicing for the computation of windows. Traub et al. [2021] provide different evaluations based on the throughput. They also assess the scalability of Scotty and conclude that it scales linear with the number of CPU cores in their tested application. However, they compare Scotty only to Flink and use throughput to evaluate the scalability. Providing further scalability benchmarks allows users to compare the scalability of an SPE with and without Scotty and lets users draw conclusions whether it is worth implementing the functionality with Scotty.

The task samples from Theodolite are derived from the Titan Control Center and are assumed to occur in other application domains as well [Henning and Hasselbring 2021d]. Theodolite’s UC3 uses hopping window aggregations. Thus, it provides a good base to implement it with Scotty and to get generally applicable results for applications that perform windowed aggregations based on time attributes.

3. Application Benchmark

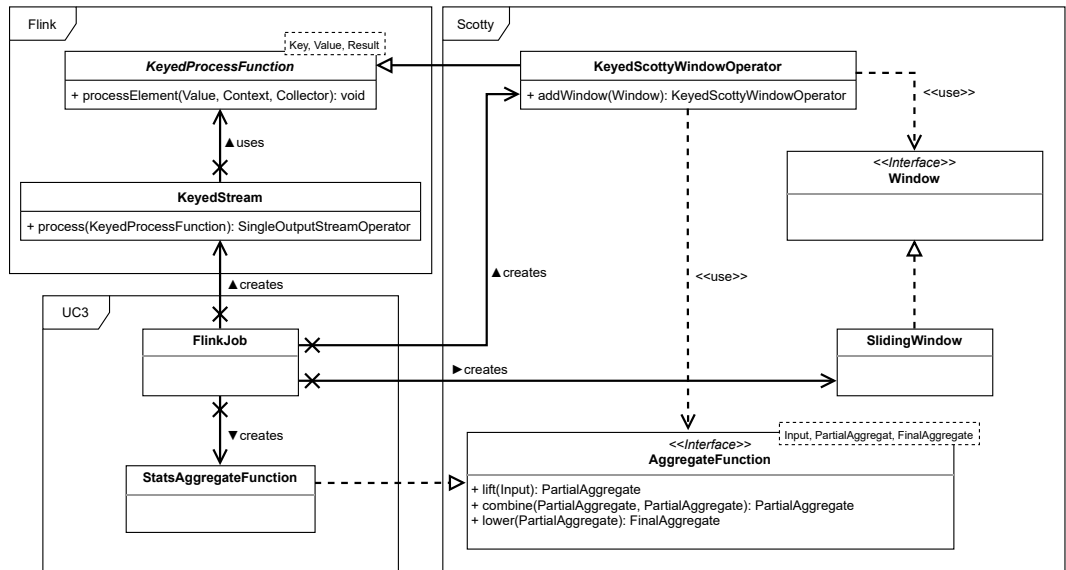


Figure 3.1. Class diagram for our implementation with Flink and Scotty.

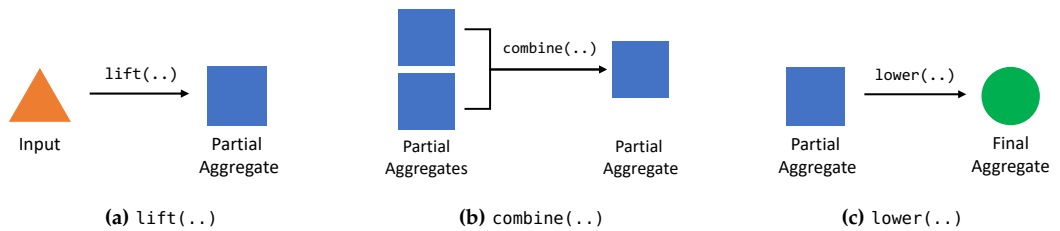


Figure 3.2. Semantics of the `lift(..)`, `combine(..)`, and `lower(..)` methods from the `AggregateFunction` interface.

3.2 Flink with Scotty

In this section, we present the implementation of UC3 with Flink and Scotty. The Scotty Framework provides some classes for the implementation with Flink. Figure 3.1 shows the classes involved in our implementation. These classes are from Flink, Scotty, and the UC3 implementation. The `KeyedScottyWindowOperator` (in the following operator) is the connector class between Scotty and Flink [Traub et al. 2021]. The operator uses an aggregation function and an arbitrary number of windows. An aggregation function needs to implement the interface `AggregateFunction` of Scotty. This requires an implementation of the three methods `lift(..)`, `combine(..)`, and `lower(..)`. Figure 3.2 shows the semantics of these functions. The `lift(..)` method creates a partial aggregate object from an input tuple

Listing 3.1. Original implementation of the topology in the FlinkJob class

```

1 // Streaming topology
2 this.env
3   .addSource(kafkaSource).name("[Kafka Consumer] Topic: " + inputTopic)
4   .keyBy((KeySelector<ActivePowerRecord, HourOfDayKey>) record -> {
5     final Instant instant = Instant.ofEpochMilli(record.getTimestamp());
6     final LocalDateTime dateTime = LocalDateTime.ofInstant(instant, timeZone);
7     return keyFactory.createKey(record.getIdentifier(), dateTime);
8   })
9   .window(SlidingEventTimeWindows.of(aggregationDuration, aggregationAdvance))
10  .aggregate(new StatsAggregateFunction(), new HourOfDayProcessWindowFunction())
11  ...

```

Listing 3.2. Scotty configuration in the FlinkJob class

```

1 final KeyedScottyWindowOperator<HourOfDayKey, ActivePowerRecord, KeyAndStats>
   processingFunction = new KeyedScottyWindowOperator<>(new
   StatsAggregateFunction());
2 final SlidingWindow slidingWindow = new SlidingWindow(WindowMeasure.Time,
   aggregationDuration.toMilliseconds(), aggregationAdvance.toMilliseconds());
3 processingFunction.addWindow(slidingWindow);

```

(Figure 3.2a). With the `combine(..)` function, a partial aggregate is computed from two partial aggregates (Figure 3.2b). Finally, with `lower(..)`, a final aggregate is computed from a partial aggregate (Figure 3.2c). Scotty offers tumbling, hopping (the `SlidingWindow` class in Figure 3.1), and session windows, among others. One or more windows can be added to the operator. In order to integrate the `KeyedScottyWindowOperator` into Flink, it extends the abstract class `KeyedProcessFunction` from Flink to become an operator (see Figure 3.1). Thus, it can be used in the *Data Stream API* as a processing step.

The task sample UC3 is specification-based and is described by a dataflow architecture (see Figure 2.12). The original Flink implementation is shown in Listing 3.1. Line 3 adds the input stream. The key is selected from Line 4 to Line 8. To use hopping windows in Flink, the `SlidingEventTimeWindows` class is used (Line 9). The window is created with a window size (`aggregationDuration`) and a sliding period (`aggregationAdvance`) that can be configured by an `application.properties` file or environment variables. After defining the window, `aggregate(..)` takes the aggregation function (Line 10) for the computation.

The code for the configuration of Scotty is provided in Listing 3.2. In Line 1 we create the Flink operator with the aggregation function. We implement the `AggregateFunction` interface in the `StatsAggregateFunction` class (Figure 3.1) and provide the same functionality

3. Application Benchmark

Listing 3.3. Original topology implementation in the TopologyBuilder class

```
1 // Streaming topology
2 this.builder
3   .stream(this.inputTopic, Consumed.with(Serdes.String(), this.srAvroSerdeFactory
4     .<ActivePowerRecord>forValues()))
5   .selectKey((key, value) -> {
6     final Instant instant = Instant.ofEpochMilli(value.getTimestamp());
7     final LocalDateTime dateTime = LocalDateTime.ofInstant(instant, this.zone);
8     return keyFactory.createKey(value.getIdentifier(), dateTime);
9   })
10  .groupByKey(Grouped.with(keySerde, this.srAvroSerdeFactory.forValues()))
11  .windowedBy(TimeWindows.of(this.aggregationDuration).advanceBy(this.
12    aggregationAdvance))
13  .aggregate(
14    () -> Stats.of(),
15    (k, record, stats) -> StatsFactory.accumulate(stats, record.getValueInW()),
16    Materialized.with(keySerde, GenericSerde.from(Stats::toByteArray, Stats::
17      fromByteArray)))
18  ...
```

as in the original aggregation. In Line 2, we create the hopping window and configure it with a configurable window size and sliding period. We add the window to the operator in Line 3. The topology for processing with Scotty is almost identical to that of Listing 3.1. Instead of the `window(..)` and `aggregate(..)` functions, we use the `process(..)` function and add the Scotty operator to it. The process function returns a stream that contains the results from the windowed aggregations. However, the key is not returned in the stream. Therefore, we also keep the key in the aggregation function and return it besides the aggregation result to provide the same results at the sink.

3.3 Kafka Streams with Scotty

In this section, we present the implementation of UC3 with Kafka Streams and Scotty. Listing 3.3 shows the original implementation of UC3 with Kafka Streams. Line 3 defines the input stream. From Line 4 to Line 8, the new key is selected based on time attributes. To perform stateful operations like window or aggregate, the stream needs to be grouped. Thus, the stream is grouped by the new selected key in Line 9. Line 10 defines the hopping window with a configured window size and sliding period. From Line 11 to Line 14, the aggregation on the windowed stream is defined.

3.3. Kafka Streams with Scotty

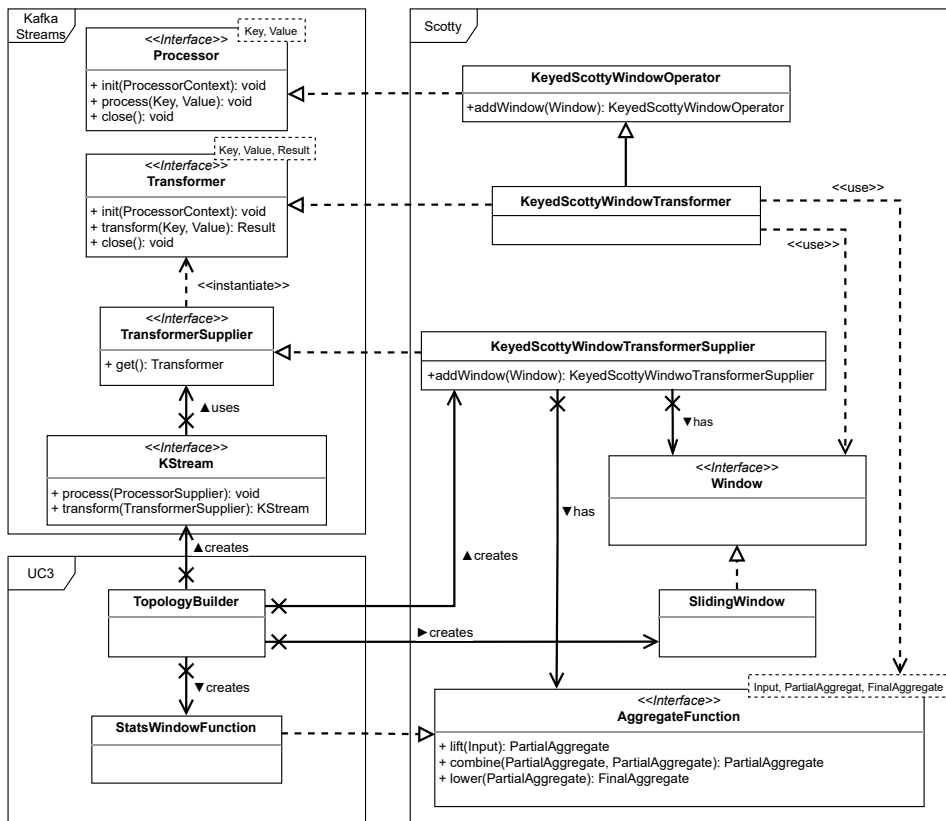


Figure 3.3. Class diagram for our implementation with Kafka Streams and Scotty.

Figure 3.3 shows the involved classes in our implementation. These classes are from Kafka Streams, Scotty, and the UC3 implementation. The Scotty framework provides a `KeyedScottyWindowOperator` class which implements the `Processor` interface of Kafka Streams. It can be used in the *Streams DSL* with the `process(..)` method of the `KStream` interface. The `process(..)` function requires a `ProcessorSupplier` for the creation of `KeyedScottyWindowOperator` objects. However, `process(..)` is a terminating function and further computations on the stream, which we need to perform, are not possible. Though, the `KStream` interface provides a `transform(..)` method that is not terminating and returns a `KStream`. The `transform(..)` function requires as an argument a `TransformerSupplier` that creates `Transformer` objects. Therefore, we create a `KeyedScottyWindowTransformer` class that implements the `Transformer` interface. The `Processor` and `Transformer` interfaces are similar and require an `init(..)` and `close()` method. However, the `Processor` interface requires a `process(..)` and the `Transformer` a `transform(..)` method. The functionality of the `KeyedScottyWindowOperator` and the new `KeyedScottyWindowTransformer` are nearly the

3. Application Benchmark

Listing 3.4. Implementation of the topology with Scotty in the TopologyBuilder class

```
1 // Scotty configuration
2 final KeyedScottyWindowTransformerSupplier<HourOfDayKey, ActivePowerRecord,
    KeyValue<HourOfDayKey, Stats>> scottyTransformerSupplier = new
    KeyedScottyWindowTransformerSupplier<>(new StatsWindowFunction(),
    GRACE_IN_SECONDS);
3 final SlidingWindow slidingWindow = new SlidingWindow(WindowMeasure.Time, this.
    aggregationDuration.toMillis(), this.aggregationAdvance.toMillis());
4 scottyTransformerSupplier.addWindow(slidingWindow);
5
6 // Streaming topology
7 this.builder
8     .stream(this.inputTopic, Consumed.with(Serdes.String(), this.srAvroSerdeFactory
        .<ActivePowerRecord>forValues()))
9     .selectKey((key, value) -> {
10         final Instant instant = Instant.ofEpochMilli(value.getTimestamp());
11         final LocalDateTime dateTime = LocalDateTime.ofInstant(instant, this.zone);
12         return keyFactory.createKey(value.getIdentifier(), dateTime);
13     })
14     .repartition(Repartitioned.with(keySerde, this.srAvroSerdeFactory.forValues()))
15     .transform(scottyTransformerSupplier)
16     .map((key, stats) -> KeyValue.pair(
17         keyFactory.getSensorId(key),
18         stats.toString()))
19     .to(this.outputTopic, Produced.with(Serdes.String(), Serdes.String()));
```

same. Therefore, the `KeyedScottyWindowTransformer` extends the `KeyedScottyWindowOperator` and reuses the `process(..)` method in the `transform(..)` method. Furthermore, we create the `KeyedScottyWindowTransformerSupplier` class that implements the `TransformerSupplier` interface. We create a pull request¹ with the changes to provide the new functionality in Scotty.

Listing 3.4 shows the new implementation with Scotty. We create the transformer supplier with the aggregation function in Line 2. We implement the `AggregateFunction` interface in the `StatsWindowFunction` class (Figure 3.3) and provide the same functionality as in the original aggregation. Further, we create the hopping window with the same window size and sliding period as the original implementation (Line 3) and add it to the supplier (Line 4). We can reuse the input topic definition and the selection of the key from the original implementation (Line 8 to Line 13). We perform a `repartition(..)` (Line 14) before we call `transform(..)` since the keys are changed. This is not mandatory because

¹<https://github.com/TU-Berlin-DIMA/scotty-window-processor/pull/39>

3.4. Kafka Streams with Sliding Window

Listing 3.5. Implementation of the topology with sliding windows in the `TopologyBuilder` class

```
1 // Streaming topology
2 ...
3 .groupByKey(Grouped.with(keySerde, this.srAvroSerdeFactory.forValues()))
4 .windowedBy(SlidingWindows.withTimeDifferenceAndGrace(this.aggregationDuration,
5     Duration.ofSeconds(GRACE_IN_SECONDS)))
6 .aggregate(
7     () -> Stats.of(),
8     (k, record, stats) -> StatsFactory.accumulate(stats, record.getValueInW()),
9     Materialized.with(keySerde, GenericSerde.from(Stats::toByteArray, Stats::
10     fromByteArray)))
11 .suppress(Suppressed.untilWindowCloses(BufferConfig.unbounded()))
12 .toStream()
13 ...
```

the time attributes are added to the original keys, and the same keys should be on the same partition. However, this is closer to the original implementation. In Line 15, we add the supplier to the `transform(..)` method. After the transformation we perform a map to have the stream in the required output data format.

Moreover, we create a bug fix² for Scotty. The `KeyedScottyWindowOperator` has a bug in its implementation and assigns wrong keys to window results. While processing tuples, the operator checks if windows are finished. If windows are finished, the operator computes the result. Then, the results get the key of the processed tuple. However, these can be windows of other keys, and, therefore, the wrong key is set to the result. With our fix, the correct key is set to the window results.

3.4 Kafka Streams with Sliding Window

This section covers the implementation of the sliding window of Kafka Streams into UC3. Listing 3.5 shows the topology implementation with a sliding window. We omit the definition of the input stream, selection of the key, and the output since this is equal to the original implementation. To use sliding instead of hopping windows, the windowing needs to be changed. Instead of a hopping window, we define a `SlidingWindows` operator with the time difference being the window size (Line 4). We do not need to define an advance time since the sliding window continuously slides over the time axis. Further, we suppress the results of the sliding windows until a final value is computed (Line 9). This means only the final value of a sliding window is published to Kafka.

²<https://github.com/TU-Berlin-DIMA/scotty-window-processor/pull/40>

3. Application Benchmark

3.5 Integration into Theodolite

The new implementations of UC3 can be integrated into Theodolite either as separate applications or as a configuration option in the current implementations. Separate applications lead to much duplicated code and, thus, we choose the configuration option. A configuration for the application is either provided through an `application.properties` file, which is contained within the application, or an environment variable. The `application.properties` file is contained in the packaged application and, therefore, needs to be set prior to the build to the desired window processor. Thus, different applications can be built and put into different docker images. The desired image can then be used for the execution. We prefer to set an environment variable during the benchmark. The environment variable is used to switch between the different window implementations in the application.

Microbenchmark

In this chapter, we accomplish goal G2 and select a microbenchmark and integrate it into Theodolite. In Section 4.1, we describe why we choose the Open Stream Processing Benchmark (OSP_{Bench}) and Section 4.2 gives an overview over OSP_{Bench}. We explain how we integrate the benchmark into the Theodolite framework for scalability benchmarking in Section 4.3.

4.1 Selection

We examine existing stream processing benchmarks in our selection. The benchmarks we take into consideration are listed in Table 4.1. In the table, we define criteria on which we base our selection for the benchmark.

Description of the Considered Benchmarks

We give a short overview of the different benchmarks. The Linear Road [Arasu et al. 2004] benchmark specifies a tolling system and compares a relational database with the SPE Aurora. In this streaming application, Aurora can outperform the database by at least a factor of 5. In Nexmark [Tucker et al. 2010], the XMark benchmark is extended with a data stream application. The scenario is an auction system and Tucker et al. [2010] define queries with different data stream operations. StreamBench [Lu et al. 2014] provides 4 different types of workloads and 7 benchmark applications. The used data is based on search data and internet traces. SparkBench [Li et al. 2015] is a benchmark suit for Spark. It provides applications from different categories like machine learning and stream processing. In the Yahoo Streaming Benchmarks (YSB), Chintapalli et al. [2016] simulate an advertisement analytics pipeline. Flink, Storm, and Spark Streaming are used to implement the pipeline. Furthermore, the performance of these engines are compared with latency and throughput. Karakaya et al. [2017] extend YSB such that it can run with multiple instances. They measure the resource usage and scalability of the frameworks with different cluster sizes. RIoT_{Bench} [Shukla et al. 2017] is a benchmark suite for real time IoT benchmarks. They provide 27 microbenchmarks and 4 application benchmarks. Karimov et al. [2018] measure the throughput and latency of windowed operations for Apache Storm, Apache Spark, and Apache Flink. The workloads and queries are based on use cases inspired by

4. Microbenchmark

the online gaming industry. Shahverdi et al. [2019] use YSB but they benchmark newer versions and additionally implement the advertisement analytics pipeline in Kafka Streams and Hazelcast Jet. They also consider resource consumption in addition to throughput and latency, which YSB evaluates. OSPBench [van Dongen and Van den Poel 2020; 2021, a; b] is a benchmark that offers different workloads and metrics for scaling efficiency, latency, throughput, and more. Various application pipelines are provided that process car traffic data. DSPBench [Bordin et al. 2020] provides 15 benchmark applications with workloads from different areas like finance, telecommunications, or sensor networks. The applications are implemented using Apache Storm and Spark Streaming, and three of the applications are compared in terms of latency, throughput, and resource usage. ESPBench [Hesse et al. 2021] is a benchmark for enterprise stream processing. It provides 5 different benchmark queries that cover the different SPE core operations. Theodolite [Henning and Hasselbring 2021d] is a method accompanied by an implementation for benchmarking the scalability of microservices and their employed stream processing frameworks. Henning and Hasselbring [2021d] provide 4 benchmark applications derived from an IIoT application (further information in Section 2.9).

4.1.1 Selection Criteria

We describe the different criteria from Table 4.1 and discuss the relevance to us.

Open Source

Open source is a desirable attribute for a benchmark [Ralph 2021]. Since we want to integrate the benchmark into Theodolite, it is helpful whether the used code is open source. If we cannot directly integrate the benchmark, we can modify the code. Furthermore, the source code of the benchmarks helps us to analyze the scalability. Most of the benchmark implementations are open source. For Linear Road, we only find an open source implementation of the data generator but not the implementation of the SUT. The authors of StreamBench [Lu et al. 2014] planned to release their benchmark on GitHub. However, we could not find it anywhere. Karakaya et al. [2017] modify YSB to work in a multi node environment. These modifications are not available as open source. For the benchmark from Karimov et al. [2018], we did not find an official implementation. However, a reproduction of the experiments is available as open source.¹ All other benchmarks provide open source implementations.

Windowed Aggregation

We study the scalability of windowed aggregations, thus, it is essential that the task sample contains windowed aggregations. The Linear Road paper [Arasu et al. 2004] indicates that windowed aggregations are used, but it is not explicitly mentioned and the source code is

¹<https://github.com/Majeux/dps1>

Table 4.1. Microbenchmark selection criteria.

Benchmark	Open Source	Window Aggr.	Type	Kafka	Stream Processing Engine
Linear Road [Arasu et al. 2004]	No	No	App	No	Aurora
Nexmark [Tucker et al. 2010]	Yes	Yes	Micro	Yes	Beam
StreamBench [Lu et al. 2014]	No	No	Micro	Yes	Spark, Storm
SparkBench [Li et al. 2015]	Yes	Yes	App	No	Spark
YSB [Chintapalli et al. 2016]	Yes	Yes	App	Yes	Flink, Spark, Storm
Karakaya et al. [2017]	No	Yes	App	Yes	Flink, Spark, Storm
RIoTBench [Shukla et al. 2017]	Yes	Yes	Mixed	No	Storm
Karimov et al. [2018]	No	Yes	Micro	No	Flink, Spark, Storm
Shahverdi et al. [2019]	Yes	Yes	App	Yes	Hazelcast Jet, Kafka Streams, Flink, Spark, Storm
OSPBench [van Dongen and Van den Poel 2020; 2021, a; b]	Yes	Yes	Mixed	Yes	Kafka Streams, Flink, Spark
DSPBench [Bordin et al. 2020]	Yes	Yes	Mixed	Yes	Spark, Storm
ESPBench [Hesse et al. 2021]	Yes	Yes	Mixed	Yes	Beam
Theodolite [Henning and Hasselbring 2021d]	Yes	Yes	App	Yes	Kafka Streams, Flink

not available to verify it. StreamBench provides count and statistics applications that can potentially use windowing. However, they plan to include windowing as future work and, therefore, it is not yet included in the benchmark. The YSB [Chintapalli et al. 2016] also uses

4. Microbenchmark

windowing. However, they do not use the window implementations of the frameworks, but their own implementation. This also applies to the benchmarks that are based on YSB.

Benchmark Type

We distinguish three benchmark types which are application benchmarks, microbenchmarks, and mixed benchmarks that contain both types of benchmarks. The distinction between application benchmark and microbenchmark is not sharp. In Section 2.8, we described that microbenchmarks measure the performance of small code fragments, a single component or task. Therefore, in this context, we consider a microbenchmark to be a stream processing application that uses only one stream operator with any required predecessor operators. For example, for many stream operations it is required to parse the ingested data first and then execute an operation on it.

To evaluate the scalability of the window aggregation, a microbenchmark is most appropriate. No operations influence the execution, and we benchmark the bare windowed aggregation. On the other side, a common application with windowing delivers realistic results for real world applications.

Kafka

Theodolites uses Kafka as the default messaging system for the benchmarks. The record lag metric is based on the record lag of Kafka. Changing the message system and scraping new metrics adds significant additional implementation effort. Thus, the load generator and SUTs should also use Kafka.

Stream Processing Engine

We study the scalability of different SPEs. Therefore, the same task sample should be implemented with different SPEs to compare the scalability. Further, these SPEs should be modern SPEs that are widely used in research and industry. With the exception of Aurora, the SPEs are widely used.

Configurable Load Generator

We survey if the benchmarks provide a configurable load generator. With a configurable load generator, we mean the benchmark has a load generator whose load we can configure. The scalability metrics of Theodolite are based on two attributes: (1) load intensity and (2) provisioned resources. In order to provide useful results, these attributes need to be configured with a set of values. Thus, the load generator needs to be able to generate the different defined loads. All of the benchmarks provide a configurable load generator. Hence, we do not explicit list this in the table.

4.2. The Open Stream Processing Benchmark (OSPBenCh)

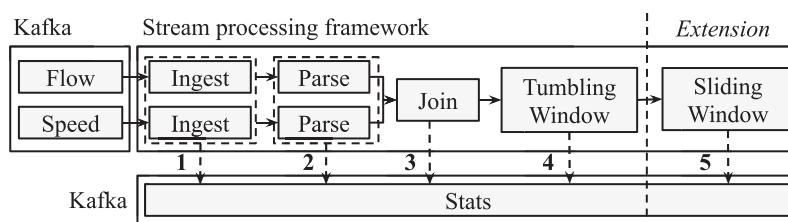


Figure 4.1. General processing pipeline of OSPBenCh [van Dongen and Van den Poel 2020].

4.1.2 Benchmark Selection

Our primary goal is to empirically evaluate the scalability of hopping window aggregations and, hence, it is essential for the benchmark to contain them. StreamBench [Lu et al. 2014] does not contain any windowed aggregation and, in consequence, is not relevant for us. YSB [Chintapalli et al. 2016] implements its own windowing function. However, we want to gain general insights into scalability of windowed aggregations and, therefore, it does not fit. Since it is important for us that the benchmark uses Kafka as the messaging system, Linear Road, SparkBench, RIoTBenCh, and the benchmark from Karimov et al. [2018] are no potential candidates. It is helpful for the analysis when the benchmark is open source. Therefore, the benchmark from Karakaya et al. [2017] is also not suitable.

From the remaining benchmarks, OSPBenCh is the only benchmark that provide a microbenchmark with hopping window aggregations. So, we decide to integrate the load generator and benchmark applications of OSPBenCh into Theodolite.

4.2 The Open Stream Processing Benchmark (OSPBenCh)

In this section, we describe OSPBenCh²[van Dongen and Van den Poel 2020; 2021, a; b] in detail. OSPBenCh uses two types of road traffic sensor data. One is the number of cars (flow) and the other the average speed of the passing cars (speed). Each record contains a measurement ID and a lane ID for identification, as well as other fields with data. The data generator sends the data in a JSON format to Kafka, and the data rate can be configured.

Different pipelines are defined that analyze the data [van Dongen and Van den Poel 2021b]. The pipelines are implemented with Flink, Kafka Streams, Spark Streaming, and Structured Streaming. In Figure 4.1, the general pipeline is shown. Stage 1 ingests the data and stage 2 parses the data into Scala objects. In the join stage both of the streams are joined. The tumbling window stage computes the average speed and total count of data with the same measurement ID. Lastly, the hopping window stage computes the relative change in flow and speed. It is possible to execute only parts of the pipeline by defining the

²<https://github.com/Klarrio/open-stream-processing-benchmark>

4. Microbenchmark

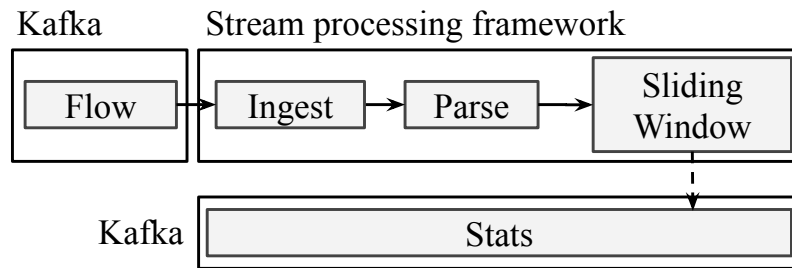


Figure 4.2. Aggregation processing pipeline of OSPBench [van Dongen and Van den Poel 2021b].

last stage. The pipeline is then executed up to this stage and contains all previous stages. A publish stage that outputs the processing results to Kafka is always added after the last stage.

Figure 4.2 shows an aggregation pipeline. This pipeline ingests, parses, and performs a hopping window aggregation on one data stream (flow) [van Dongen and Van den Poel 2021b]. The hopping aggregation has a default window size of 5 minutes and a sliding period of 1 minute. It accumulates the count of the passed cars for the same measurement ID. Results of the aggregation are written back into a Kafka topic.

OSPBench has metrics for latency, throughput, memory and CPU utilization, garbage collection (GC), network utilization, and filesystem and disk I/O [van Dongen and Van den Poel 2020; 2021, b]. Furthermore, workloads for latency, sustainable throughput, burst at startup, periodic bursts, and scalability measurements are defined. Therefore, different types of load generators are implemented. The components of the benchmark run in Docker containers on DC/OS.³

4.3 OSPBench Integration into Theodolite

In the following, we explain the different steps to integrate OSPBench into Theodolite and the modifications we apply to OSPBench. The modifications we apply to OSPBench can be found at GitHub.⁴ The pipeline we integrate into Theodolite is the aggregation pipeline (Figure 4.2).

4.3.1 Workload Generator Integration into Theodolite

We start by integrating the workload generator. OSPBench already provides a container image for its workload generator [van Dongen 2021]. The workload generator has a `resource.conf` file which contains configuration options. This configuration file is loaded with the

³<https://dcos.io>

⁴<https://github.com/bvonheid/open-stream-processing-benchmark>

4.3. OSPBench Integration into Theodolite

Listing 4.1. Dockerfile for the OSPBench workload generator

```
1 FROM openjdk:8-slim
2
3 # data to ingest
4 COPY src/main/resources/data/ /data
5 ENV LOCAL_PATH=/data/time*/
6
7 # executable
8 ADD target/scala-2.11/ospbench-data-stream-generator-assembly-3.0.jar /
9
10 # execute
11 ENTRYPOINT ["java", "-jar", "ospbench-data-stream-generator-assembly-3.0.jar"]
```

Lightbend Configuration library.⁵ In the configuration, we can set if the sensor data is either loaded from a local file system or from Amazon S3. Further, the type of workload, the pipeline, and the load capacity can be set. Multiple instances of the workload generator can be started to generate higher loads. Therefore, each load generator should have a unique ID so that the generated keys are different. Furthermore, the Kafka bootstrap server and the topics can be set.

We want the load generator to load sensor data from a local file rather than from Amazon S3 since this would create an additional infrastructure component in Theodolite. Though, the provided Docker image does not contain the data and can only load the data from Amazon S3. One solution is to provide the data through a volume to the running container. However, the volume needs to be created in the process of the benchmark and, hence, adds additional complexity to the definition of the benchmark. The solution we choose is to include the data directly into the container image of the load generator. Therefore, we create a new Dockerfile shown in Listing 4.1. In Line 4, the sensor data is added to image and Line 5 sets the path for the program to load the data.

We create a Kubernetes deployment resource (Listing 4.2) for the deployment of the workload generator in Kubernetes. We use the Docker image (Line 14) created with the Dockerfile above (Listing 4.1) for the workload generator. From Line 15 to Line 21, we define the address for the Kafka server and set the ID for the load generator. As the load generator ID, we use the uid of the pod which is unique. The resources provided for the load generator (Line 22 to Line 25) are the same as those set by OSPBench in their experiments. Moreover, we set `terminationGracePeriodSeconds` to 0 (Line 11). Thus, the pod is directly removed when the experiment is finished since we do not need the pod to shut down gracefully. This reduces the total time we need to execute the benchmark.

⁵<https://github.com/Lightbend/config>

4. Microbenchmark

Listing 4.2. Kubernetes deployment for the OSPBench workload generator [Vonheiden 2021]

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: osp-load-generator
5 spec:
6   ...
7   replicas: 1
8   template:
9     ...
10    spec:
11      terminationGracePeriodSeconds: 0
12      containers:
13        - name: workload-generator
14          image: ghcr.io/bvonheid/ospbench-data-stream-generator:bv-thesis-1
15          env:
16            - name: KAFKA_BOOTSTRAP_SERVERS
17              value: "theodolite-cp-kafka:9092"
18            - name: PUBLISHER_NB
19              valueFrom:
20                fieldRef:
21                  fieldPath: metadata.uid
22      resources:
23        limits:
24          memory: 6Gi
25          cpu: 3000m
```

We deployed the load generator to Kubernetes and tested it with different data volumes. *Data volume* is the load intensity used in the load generator and defines how many messages per second the generator produces. The number of different keys is equal to the number of messages per second. Regarding van Dongen [2021], the *throughput* in messages per second published to Kafka is for a single stream:

$$\text{throughput} = 190 \times (\text{data volume} + 1)$$

However, we observed the following rates shown in Table 4.2, printed to the console by the publisher and also scraped by Prometheus. The data volume does not scale linearly and we have the same throughput for different data volumes. The problem is the following line of the workload generator:

```
0.to(Math.round(ConfigUtils.dataVolume.toInt/3.0).toInt).foreach {...}
```


4.3. OSPBench Integration into Theodolite

Table 4.2. Generated throughput in messages/sec for different data volumes.

Data Volume	Throughput
0	570
1	570
2	1140
3	1140
4	1140
5	1710

The given data volume is divided by three and the result is rounded and converted to Int. For example, dividing two, three, and four by this formula yields the same result and, therefore, the same throughput. We change this line to

```
1.to(ConfigUtils.dataVolume).foreach {...}
```

which leads to a ratio scale data volume. With a data volume of 0, no data is produced and the data volume 1 is the lowest value for generating data. The formula for throughput is now:

$$\text{throughput} = 570 \times \text{data volume}$$

We create a pull request⁶ on GitHub to fix this issue.

Next, we integrate the SUTs with Kafka Streams, Flink, and Spark into Theodolite. The SUTs are configured with different configuration files. A `commonsettings.conf` file defines settings that apply across the applications. Furthermore, there are the `local.conf`, `docker.conf`, and `aws.conf` files. One of them need to be selected via an environment variable. They contain common settings like which pipeline to execute and configurations for the different platforms. After all, each SUT contains a further configuration file which configures the SPEs based on the workload. Some of the configuration settings, but not all, can be modified through environment variables.

4.3.2 Kafka Streams SUT Integration into Theodolite

We make the commit interval of the Kafka Streams SUT configurable over an environment variable. The commit interval defines how often the offset of topics should be saved in Kafka [Apache Software Foundation 2021c]. In OSPBench, the commit interval for the Kafka Streams application is based on the chosen pipeline and is set to 60 seconds for the aggregation pipeline. A commit interval of 60 seconds can be used in Theodolite, but it requires a longer execution time to get accurate results. Theodolite uses the record lag SLO and, thus, depends on the record lag of Kafka. The record lag in Kafka is computed by the number of produced messages and the latest offset from the consumed messages.

⁶<https://github.com/Klarrio/open-stream-processing-benchmark/pull/5>

4. Microbenchmark

Listing 4.3. Theodolite benchmark definition for the OSPBench Kafka Streams SUT [Vonheiden 2021]

```
1 apiVersion: theodolite.com/v1
2 kind: benchmark
3 metadata:
4   name: osp-kstreams
5 spec:
6   appResource:
7     - "custom/osp-kstreams-deployment.yaml"
8     - "custom/osp-kstreams-service.yaml"
9     - "custom/osp-service-monitor.yaml"
10  loadGenResource:
11    - "custom/osp-load-generator-deployment.yaml"
12  resourceTypes:
13    - typeName: "Instances"
14      patchers:
15        - type: "ReplicaPatcher"
16          resource: "custom/osp-kstreams-deployment.yaml"
17  loadTypes:
18    - typeName: "TotalDataVolume"
19      patchers:
20        - type: DataVolumeLoadGeneratorReplicaPatcher
21          resource: "custom/osp-load-generator-deployment.yaml"
22        properties:
23          maxVolume: "400"
24          container: "workload-generator"
25          variableName: "DATA_VOLUME"
26  kafkaConfig:
27    bootstrapServer: "theodolite-cp-kafka:9092"
28    topics:
29      ...
```

To execute a benchmark with Theodolite, the user must define a benchmark and an execution. In the following, we show the general definitions of benchmarks and executions in Theodolite. We go into more detail about the exact configurations in Chapter 5.

The benchmark definition for the Kafka Streams SUT is shown in Listing 4.3. A benchmark requires resources for the SUT (Line 6 to Line 9) and the load generator (Line 11). Resources are Kubernetes resource files that, for example, define deployments or services. The resourceType (Line 12) defines the available types for provisioned resources. We specify the number of instances as the resource type. It configures how many pods with the Kafka Streams application are started in Kubernetes. The loadTypes (Line 17) defines the available load intensities. As the load type, we define a total data volume generated by one or more

4.3. OSPBench Integration into Theodolite

Listing 4.4. Theodolite example execution definition for the OSPBench Kafka Streams SUT [Vonheiden 2021]

```
1 apiVersion: theodolite.com/v1
2 kind: execution
3 metadata:
4   name: osp-kstreams-default
5 spec:
6   benchmark: "osp-kstreams"
7   load:
8     loadType: "TotalDataVolume"
9     loadValues: [5, 10, 15, 20, 30, 40, 50, 60]
10  resources:
11    resourceType: "Instances"
12    resourceValues: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 16, 18, 20]
13  slo:
14    - sloType: "lag trend ratio"
15    ...
16  execution:
17    strategy: "LinearSearch"
18    duration: 360 # in seconds
19    repetitions: 3
20    loadGenerationDelay: 60 # in seconds
21  restrictions:
22    - "LowerBound"
23  configOverrides:
24    - patcher:
25      type: "EnvVarPatcher"
26      resource: "custom/osp-kstreams-deployment.yaml"
27      properties:
28        container: "uc-application"
29        variableName: "LAST_STAGE"
30      value: "100"
31    ...
```

load generators. We create a new patcher for this and explain it in Section 5.1. Lastly, we define the Kafka address and topics (Line 26).

Listing 4.4 shows an example execution for the benchmark defined in Listing 4.3. We define a list of loads (Line 9) and a list of instances (Line 12) for which SLO experiments should be executed. At least one SLO needs to be defined for the execution (Line 13). We use the new SLO *lag trend ratio* that is introduced by us and explained in more detail in Section 5.1. In the execution section (Line 16 to Line 22), we define which search strategy

4. Microbenchmark

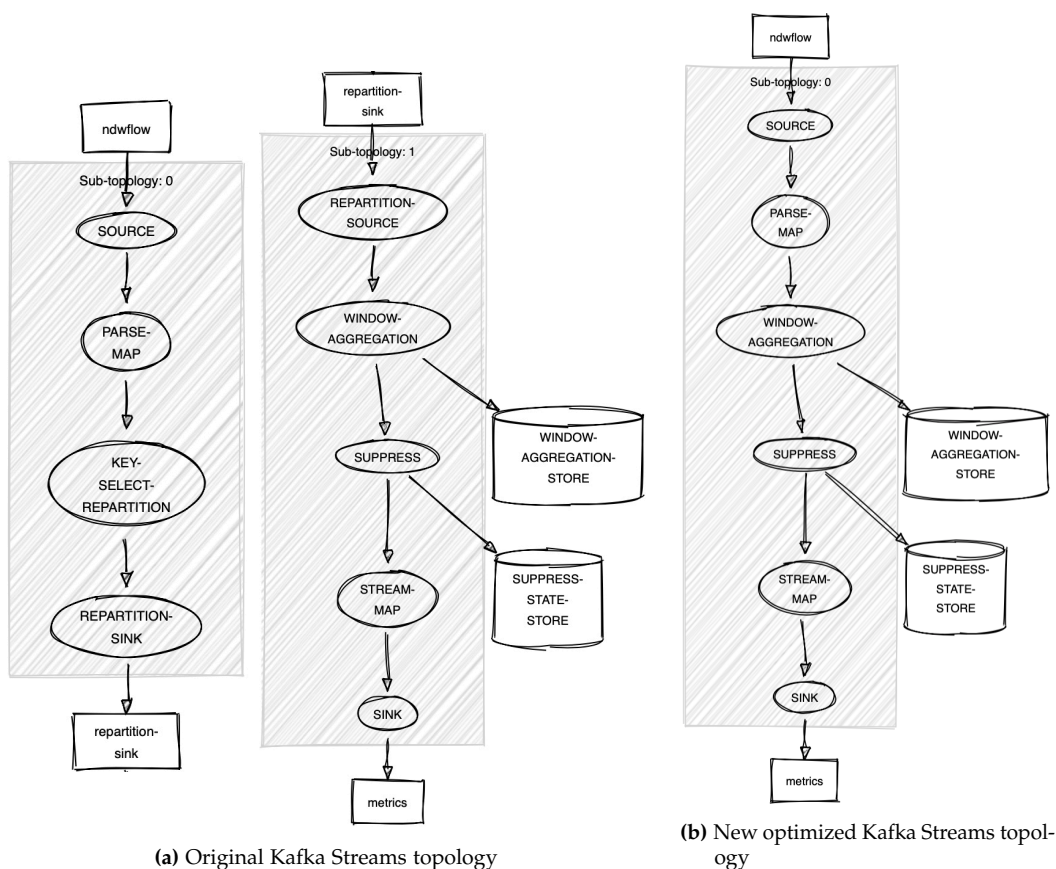


Figure 4.3. The original and the new optimized Kafka Streams topology (simplified).

should be used. Besides, the execution time for each SLO experiment, the number of repetitions of an SLO experiment, and more are defined. In the `configOverrides`, we define patchers that we can use to change Kubernetes resource files. The patcher in Line 24 sets an environment variable in the SUT to define which pipeline should be executed in the benchmark. Moreover, we configure the window size, sliding period, commit interval, and more with patchers.

During the integration, we looked at the Kafka Streams code of `OSPBench`. The topology created by this code is shown in Figure 4.3a. We simplify the topology and merge some operations in one task. Two sub-topologies are created in the implementation. The first sub-topology on the left ingests the data, parses the data, and selects the new key. The selection is performed by the `groupBy(...)` operation of Kafka Streams and uses a property of the message as the key. This operation always causes a repartition of the stream [Apache Software Foundation 2021c]. A repartition writes the data into an internal Kafka topic,

4.3. OSPBench Integration into Theodolite

Listing 4.5. Dockerfile for the OSPBench Flink SUT

```
1 FROM flink:1.11-scala_2.12-java8
2
3 ADD target/scala-2.12/flink-benchmark-assembly-3.0.jar /opt/flink/usrlib/artifacts
   /flink-benchmark.jar
```

which is in this case the repartition sink. The second sub-topology on the right reads the data from this topic and performs the windowed aggregation. However, the key that is selected by the `groupBy` operation is already the key of the message. The Kafka Streams documentation recommends using the `groupByKey` instead of the `groupBy` operation if possible [Apache Software Foundation 2021c]. With `groupByKey` a repartition of the data is only performed if the stream was already marked for repartition before. We change the code to use the `groupByKey` operation and the program produces the topology shown in Figure 4.3b. Only a sub-topology is created and not an internal topic. We provide this optimized application in an extra Docker image and also create a Theodolite execution for it.

4.3.3 Flink SUT Integration into Theodolite

In Section 2.5, we describe the components of a Flink cluster, the cluster deployment modes, and the application deployment modes. Theodolite and OSPBench create a standalone cluster for the benchmarks. We also deploy the cluster in our benchmark executions as a standalone cluster. The standalone cluster has the advantage that we are in control of the components and resources of Flink. We follow the documentation for running a standalone cluster in Kubernetes [Apache Software Foundation 2021b] for the creation of the Kubernetes resource files.

OSPBench uses session mode and Theodolite uses application mode for the application deployment. In the session mode, the Flink client is a separate component, and in the application mode, the Flink client runs directly on the job manager. With the application mode we do not need to start an additional component and, further, it reduces the CPU cycles and network bandwidth [Apache Software Foundation 2021b].

In the application mode, the application artifact needs to be available on the classpath of the job manager and task managers [Apache Software Foundation 2021b]. The artifact can either be provided via a volume or by creating a custom image containing the artifact. The provided docker images from OSPBench do not contain the artifacts and, thus, we create a Dockerfile to create an image (Listing 4.5). We use a base docker image (Line 1) that is provided by Flink and add the job artifact (Line 3). The resulting Docker image is used in the Kubernetes resource files for the job manager and task manager.

4. Microbenchmark

Listing 4.6. Dockerfile for the OSPBench Spark Streaming SUT

```
1 FROM bde2020/spark-submit:3.0.2-hadoop3.2
2
3 WORKDIR /app
4
5 ENV ENABLE_INIT_DAEMON false
6 ENV SPARK_APPLICATION_JAR_LOCATION /app/spark-benchmark-assembly-3.0.jar
7 ENV SPARK_APPLICATION_MAIN_CLASS spark.benchmark.SparkTrafficAnalyzer
8
9 COPY target/scala-2.12/spark-benchmark-assembly-3.0.jar /app
```

The benchmark and execution definitions are similar to Listing 4.3 and Listing 4.4. They differ in the app resource files that are used and in the environment variables that are set with the patchers.

4.3.4 Spark SUT Integration into Theodolite

OSPBench provides two applications that are based on Spark. One is implemented with Spark Streaming and the other with Structured Streaming. Spark applications requires a driver program, a cluster manager, and worker nodes (cf. Section 2.6). The cluster manager can either be standalone, Mesos, YARN, or Kubernetes [Apache Software Foundation 2021d]. A spark submit script (submitter) is used to start the driver program. The script provides two deployment modes for the driver program: client mode and cluster mode. In the client mode, the driver is started in the spark submit process. With the cluster mode the driver is started on one of the worker nodes.

Create Kubernetes resource files

In OSPBench the standalone cluster mode is used and the driver runs in client mode. Van Dongen and Van den Poel [2021b] use the client mode in order to fully utilize the resources of the workers. Thus, we also use the client mode. OSPBench provides Docker images for the master, worker, and submitter. However, we create our own submitter image because we modify the application and cannot set all required configuration options with the provided image. Furthermore, we use existing master and worker images.

We use the `bitnami/spark` Docker image⁷ for the master and the workers. For the Spark Streaming submitter, we create a custom Dockerfile as shown in Listing 4.6. We use a base docker image (Line 1) provided by Big Data Europe.⁸ Two environment variables set the

⁷<https://hub.docker.com/r/bitnami/spark>

⁸<https://github.com/big-data-europe>

4.3. OSPBench Integration into Theodolite

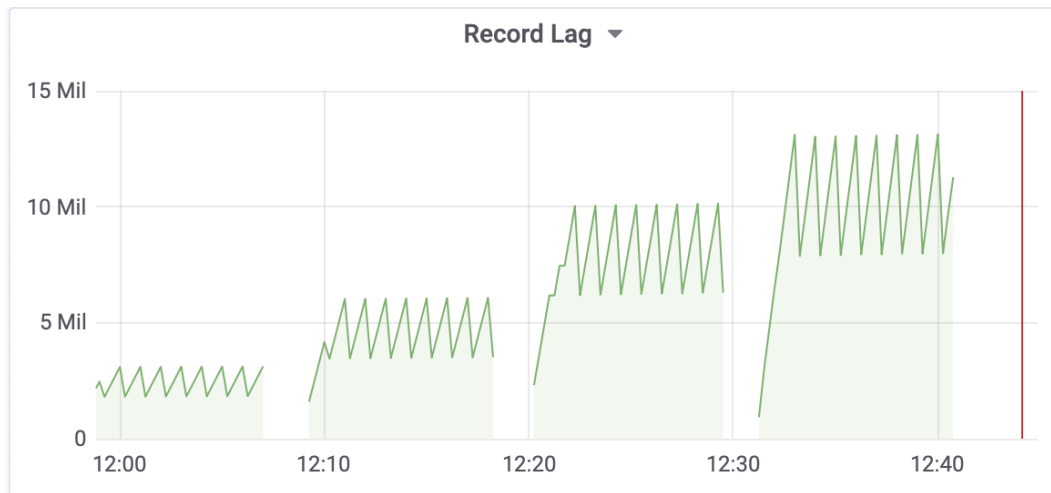


Figure 4.4. Spark Streaming record lag with auto commit.

jar (Line 6) and the main class (Line 7). In Line 9, we copy the jar into the container. The Dockerfile is similar for the Structured Streaming submitter.

We create Kubernetes deployments for the Spark master (cluster manager), worker, and submitter (driver program). Furthermore, the master and submitter need to be reachable inside Kubernetes and, thus, we create services for them. The master and worker deployments use the `bitnami/spark` image. This image can be configured with the `SPARK_MODE` environment variable to either start as a master or a worker. In the deployment for the driver, we use our submitter image, configure the environment variables, and set arguments that are passed to the submit script.

Spark offset commit

We test the Spark Streaming and Structured Streaming implementations in a Kubernetes cluster. For Spark Streaming we observe that the record lag has a repeated pattern for different loads (Figure 4.4). It has peaks at 1-minute intervals with drops in between. The record lag drops to nearly the same record lag and, thus, leads to the conclusion that the load can be processed. However, we observe in the Spark dashboard at high loads that the processing of batches is delayed and the processing time for a batch is greater than the window sliding time (Figure 4.5). Thus, the load cannot be processed.

The Spark Streaming implementation uses the auto commit function to commit offsets to Kafka. Though, when the polled data is committed, no operation may have been performed for this data [Apache Software Foundation 2021d]. We disable the auto commit function and commit the offset manually while processing. The manual commit should be executed before any transformation is performed on the stream [Apache Software Foundation 2021d].

4. Microbenchmark

Active Batches (5)

Batch Time	Records	Scheduling Delay ^(?)	Processing Time ^(?)	Output Ops: Succeeded/Total	Status
2021/11/04 21:34:00	7292200 records	-	-	0/1 (1 running)	queued
2021/11/04 21:33:00	7292200 records	-	-	0/1 (1 running)	queued
2021/11/04 21:32:00	9465040 records	-	-	0/1 (1 running)	queued
2021/11/04 21:31:00	9465038 records	-	-	0/1 (1 running)	queued
2021/11/04 21:30:00	27360945 records	4,4 min	-	0/1 (1 running)	processing

Completed Batches (last 5 out of 5)

Batch Time	Records	Scheduling Delay ^(?)	Processing Time ^(?)	Total Delay ^(?)	Output Ops: Succeeded/Total
2021/11/04 21:29:00	27359055 records	3,3 min	2,1 min	5,4 min	1/1
2021/11/04 21:28:00	27360000 records	2,2 min	2,1 min	4,3 min	1/1
2021/11/04 21:27:00	27363731 records	43 s	2,5 min	3,2 min	1/1
2021/11/04 21:26:00	9648269 records	1 ms	1,7 min	1,7 min	1/1
2021/11/04 21:25:00	0 records	10 ms	5 s	5 s	1/1

Figure 4.5. Spark Streaming batch processing time and scheduling delay.

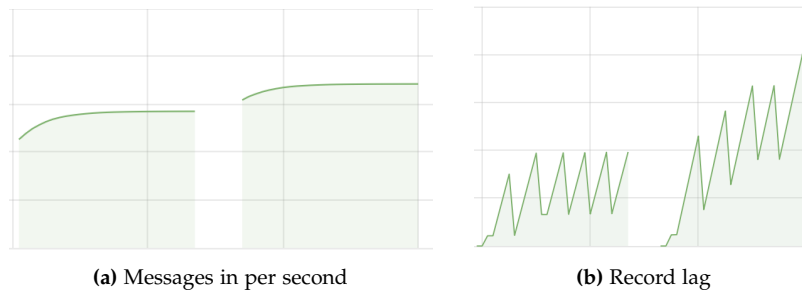


Figure 4.6. Spark Streaming with manual offset commit.

Thus, we commit the data before we execute the windowed aggregation. However, a job is created for each window and jobs are executed in a *first in first out* order by default. Therefore, the next batched data is first committed when the previous data is processed. In Figure 4.6, we show the record lag for different loads with manual commits. Two SLO experiments are executed with different loads (Figure 4.6a). In Figure 4.6b, the record lags for the SLO experiments are shown. The left record lag indicates that the load is processed by the instances. In the right SLO experiment, the load is higher and the record lag indicates that the messages are queuing up.

In Structured Streaming the auto commit is disabled by default and cannot be activated [Apache Software Foundation 2021d]. Structured Streaming uses its own mechanism with checkpointing and write-ahead logs to save checkpoints. Thus, we are not able to monitor any record lag for the Structured Streaming SUT.

4.3.5 Theodolite Integration Summary

Since it is the first time Theodolite is used to benchmark another SUT, we briefly summarize the integration of the OSPBench SUTs into Theodolite. The most time-consuming tasks for us were getting familiar with OSPBench and creating the cluster components for the Spark applications in Kubernetes. Spark only provides minimal documentation for running it as a standalone cluster in Kubernetes. Thus, we need to create the Kubernetes resource files from scratch. In contrast, we were able to reuse the resources of the Kafka Streams and Flink benchmark applications of Theodolite. Hence, the integration of Kafka Streams and Flink has been made easier for us.

The SLOs of Theodolite use the record lag metric of Kafka. As long as the SUTs reliably commit the offset, nothing needs to be adapted for the measuring. Unfortunately, Structured Streaming does not commit any offset at all and, thus, we were not able to benchmark the Structured Streaming application with Theodolite. However, Theodolite offers the possibility to define new SLOs.

We present some general steps for the integration of benchmarks into Theodolite. If the workload generator and SUT can be deployed as containers and interact with Kafka, the following steps can be used to introduce new benchmarks into Theodolite:

1. Create Kubernetes resources for workload generator and SUT
2. Define a benchmark
3. Define an execution

Experimental Evaluation

In this chapter, we accomplish goal G3 and evaluate the scalability of hopping window aggregation methods. In Section 5.1, we describe the extensions we make to Theodolite to execute our benchmarks. We explain our test setup, the window configurations we use in the benchmarks, and the configurations of the SUTs in Section 5.2. In Section 5.3, we explain the results of the benchmarks. We discuss threats to the validity of our results in Section 5.4.

5.1 Theodolite Extensions

In the Theodolite UC3 benchmarks, the load dimension is the number of sensors. The range of the load intensities vary in our benchmarks between small loads of 50 sensors and higher loads of 500 000 sensors. Currently, Theodolite provides only the *lag trend* SLO. The SLO uses the *lag trend metric* (cf. Section 2.9.1) and compares the computed lag trend to an absolute defined threshold. If the lag trend is below the defined threshold, the SLO is fulfilled. For experiments that have a broad range of loads, it is hard to determine what an acceptable threshold is. To adapt the threshold to the load that is applied, we introduce a *lag trend ratio* SLO. In contrast to the *lag trend* SLO with an absolute threshold, a ratio is defined. An absolute threshold is computed for each SLO experiment and is calculated based on the ratio and load of the SLO experiment using the following formula:

$$\text{threshold} = \text{load} \times \text{ratio}.$$

Just like the *lag trend* SLO, the resulting absolute threshold is compared to the lag trend.

The load generator of OSPBench uses the data volume as load dimension (cf. Section 4.3.1). In the executions we define a total data volume that one or more generators should generate together. Therefore, we need to compute the required number of load generators for the generation of the total data volume. Furthermore, the load generators generate the load independently and, hence, a data volume per generator need to be set. Thus, we need a patcher (cf. Section 2.9.1) in Theodolite that computes the number of required load generators and sets for the load generators the data volume they should produce.

Theodolite provides a patcher for scaling the number of load generators. The patcher is defined with a maximal load that a generator instance can generate (max load) and calculates based on a total load the required number of generators. The number of generators is

5. Experimental Evaluation

calculated with:

$$\#generators = \left\lfloor \frac{\text{total load} + \text{max load} - 1}{\text{max load}} \right\rfloor$$

This is also the semantic needed for calculating the required number of generators with OSPBench. However, the load per generator is not computed and set for the generator instances. So, we create a new patcher that calculates the number of required generators with the formula above and the data volume that should be generated by each of the generators. The data volume for each generator instance is computed using

$$\text{data volume} = \left\lfloor \frac{\text{total data volume}}{\#generators} \right\rfloor.$$

The computed data volume is set as an environment variable in the load generators.

5.2 Methodology

Hardware

We execute the benchmarks in a Kubernetes cluster¹ (version 1.18) that consists of 5 nodes. Each of the nodes has 348 GB RAM and 2×16 CPU cores. Thus, there are 160 cores in total. The nodes are connected with 10 Gbit/s Ethernet.

Theodolite Deployment

We deploy Theodolite in Kubernetes with Helm.² The deployment includes Prometheus, Grafana, and Kafka. We use 10 Kafka Brokers and for every topic in the experiment 40 partitions.

Benchmark Window Configurations

Table 5.1 shows the SUTs and window configurations we use in the benchmark executions. We execute each SUT with each window configuration to determine the influence of different window sizes, sliding periods, and number of overlapping windows. For UC3 we benchmark the original SUTs and compare them to the new SUTs with the Scotty framework and the Kafka Streams sliding window. We benchmark three different window configurations with each UC3 SUT. These are windows with a 30 days window size and 1 day sliding period, a 5 minutes window size and 1 minute sliding period, and a 30 seconds window size and 1 second sliding period. There are 30 overlapping windows with the 30 days and 30 seconds window sizes. The 30-second window has a sliding period of 1 second, thus, new windows are created and results published to Kafka more frequently.

¹<https://www.se.informatik.uni-kiel.de/en/research/software-performance-engineering-lab-spel>

²<https://helm.sh>

Table 5.1. SUTs and window configurations of our benchmark executions.

Benchmark	Systems Under Test	Window Configurations	
		window size	sliding period
UC3	Kafka Streams, Kafka Streams +	30 days	1 day
	Scotty, Kafka Streams + Sliding	5 minutes	1 minute
	Window, Flink, Flink + Scotty	30 seconds	1 second
UC3	Kafka Streams + Scotty, Flink + Scotty	20 minutes	1 minute
OSPBench	Kafka Streams, Flink, Spark Streaming	10 minutes	2 minutes
		5 minutes	1 minute
		5 minutes	15 seconds
		150 seconds	30 seconds

With the 5 minutes window size, there are 5 parallel windows. Additionally, we benchmark the Scotty SUTs with a 20 minutes window size and a sliding period of 1 minute.

Since there are no existing scalability results for the OSPBench incremental aggregation pipeline, we evaluate its scalability. Therefore, we execute the experiments with suitable load intensities. Then we can analyze and assess the scalability. For OSPBench, we benchmark the SUTs implemented with Kafka Streams, Flink, and Spark Streaming. We use three window configurations with 5 overlapping windows that have different window sizes. Moreover, we have one window configuration with 20 overlapping windows. To compare the difference in scalability of both benchmark applications, we execute all the SUTs with a window size of 5 minutes and sliding period of 1 minute.

Load Dimension and Resource Dimension

We use the number of different keys sent per second as the load dimension in both benchmarks. In UC3 we can directly define the number of different keys. The load generator of OSPBench scales with the data volume (Section 4.3.1), which indirectly defines the number of different keys. We use the number of processing instances as the resource dimension. An instance for Kafka Streams is a Kubernetes pod that contains the application. For Flink, the number of instances is equal to the number of task manager pods, and for Spark, to the number of worker node pods.

Execution Configurations

Table 5.2 shows the execution configurations we use in the SLO experiments. We execute the SLOs experiments of UC3 for 5 minutes, of OSPBench for 6 minutes, and for Spark OSPBench for 7 minutes. The workload generator of OSPBench needs slightly more time than that of UC3 to produce the defined load and, thus, we execute the OSPBench SLOs experiments

5. Experimental Evaluation

Table 5.2. Configurations for the SLO experiments. For Spark the *lag trend ratio* depends on the sliding period. The sliding periods are denoted behind the ratio.

Configuration Option	UC3	OSPBench
SLO experiment duration	5 minutes	6 minutes Spark: 7 minutes
SLO experiment warmup	60 s	60 s
SLO experiment repetitions	3	3
SLO type	<i>lag trend ratio</i>	<i>lag trend ratio</i>
<i>lag trend ratio</i>	0.05	28.5 Spark: 28.5 (s: 15 s), 45.6 (s: 30 s), 85.5 (other)
search strategy	lower bound with linear search	lower bound with linear search
Kafka partition count	40	40
Kafka Streams commit interval	100 ms	5 s
Kafka Streams Instance CPU, memory	1 core, 4 GB	1 core, 4 GB
Flink checkpointing	100 ms	5 s
Flink JobManager CPU, memory	1 core, 4 GB	2 cores, 8 GB
Flink Taskmanager CPU, memory	1 core, 4 GB	1 core, 4 GB
Spark batch interval	<i>not available</i>	5 s
Spark Master CPU, memory	<i>not available</i>	2 cores, 8 GB
Spark Worker CPU, memory	<i>not available</i>	1 core, 4 GB
Spark Submitter CPU, memory	<i>not available</i>	2 cores, 6 GB

longer. We execute the Spark OSPBench SUT for 7 minutes since Spark commits the offset of the consumed messages irregularly in Kafka. Furthermore, we apply a warmup of 1 minute to all the benchmarks, because the load generator and the SUTs need some time to start. With the warmup we ignore the record lag during the startup in the analysis. Every SLO experiment is executed with 3 repetitions to gain more confidence in the results.

5.2. Methodology

Furthermore, we use the *lag trend ratio* SLO in all experiments. For the UC3 SLO experiments we use a 5% ratio and for OSPBench a ratio of 28.5. The workload generator of OSPBench uses the data volume and not messages/second as load intensity. The ratio of 28.5 yields 5% of the generated message because:

$$\frac{\text{data volume} \times 28.5}{\text{data volume} \times 570} = \frac{28.5}{570} = 0.05 = 5\%.$$

We use different ratios for Spark. The Spark SUT commits the offset irregularly and the time of the commit depends on the sliding period. Figure 4.6 shows that the record lag with Spark increases linearly and when the SUT commits the offset, the record lag drops. We take this into account for the ratio of the SLO experiments with Spark. In the worst case, the lag trend is computed starting with a low record lag and ending with a peak. With the wrong ratio, the analysis produces false negatives and our results are inaccurate. If the Spark SUT can process the load, the maximum number of queued messages (max messages) between a low and a peak is:

$$\text{max messages} = \text{sliding period} \times \text{messages/sec}.$$

We choose the maximum number of queued messages as an acceptable lag over the whole SLO experiment. With the maximum number of queued messages, we compute the message ratio. The message ratio give us the ratio of the total number of messages/second that the record lag is allowed to increase and is computed with:

$$\text{message ratio} = \frac{\text{max messages}}{\text{duration} - \text{warmup}}.$$

For the sliding period of 15 seconds, we get a message ratio of approximately 0.05, for 30 seconds approximately 0.08, for 60 seconds approximately 0.15, and for 120 seconds approximately 0.33. With the message ratio, we can compute the ratio for the data volume:

$$\text{data volume ratio} = \text{message ratio} \times 570.$$

The computed ratio of the 120 seconds sliding period is very large, so we use the data volume ratio of the 60 seconds sliding period. The computed ratios for Spark are shown in Table 5.2.

We use the lower bound restriction with linear search as the search strategy in the SLO experiments [Henning and Hasselbring 2021a].³ The lower bound restriction reduces the search space by the assumption that a larger load needs at least as many instances as the preceding smaller load. For searching it uses one of the search strategies provided by Theodolite. Linear search assumes that if I instances can process the load, then I+1 instances can process the load. It starts for any load with the lowest number of instances. If the tested number of instances can process the load, the next load is tested, otherwise

³In a previous publication, Henning and Hasselbring [2020] call the linear search *heuristic H1* and the lower bound restriction *heuristic H3*.

5. Experimental Evaluation

the load is tested with the next higher number of instances. By using the lower bound restriction with linear search, we can reduce the total number of executed experiments.

Configurations of the Kafka Streams SUTs

In the SUTs of UC3, we use the default configs of Theodolite [Henning and Hasselbring 2021d]. For the Kafka Streams SUT of OSPBench, we set the commit interval to 5 seconds instead of 60 seconds to get more recent record lags. The Kafka Streams instances are configured with 1 CPU core and 4 GB memory.

Configurations of the Flink SUTs

In the SUTs of UC3, we use the default configs of Theodolite [Henning and Hasselbring 2021d]. Like the Kafka Streams commit interval, we set the Flink checkpoint interval of the OSPBench SUT to 5 seconds. We deploy one job manager in the Flink benchmarks. For OSPBench, the job manager is configured with 2 CPU cores and 8 GB memory as in [van Dongen and Van den Poel 2021b]. The task managers are configured with 1 CPU core, 4 GB memory, and have 1 task slot each.

Configurations of the Spark Streaming SUT

In Spark, we set the batch interval to 5 seconds instead of 60 seconds. Thus, we can set smaller sliding periods for the windows. As in [van Dongen and Van den Poel 2021b], the master is configured with 2 CPU cores and 8 GB memory, and the submitter is configured with 2 CPU cores and 6 GB memory. The worker instances are configured with 4 GB memory and 1 CPU core. For Flink and Spark, we set the parallelism to the number of task managers and worker instances, respectively.

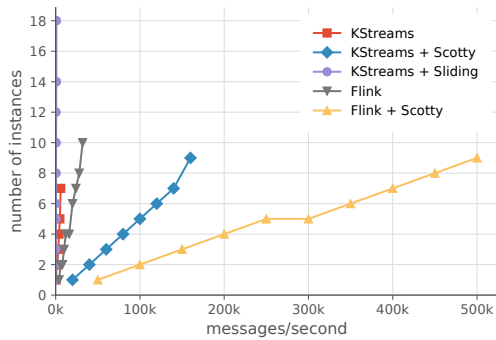
We provide a replication package [Vonheiden 2021] that contains instructions for repeating the experiments and analysis. Further, it contains the measurements of the executed experiments.

5.3 Results and Discussion

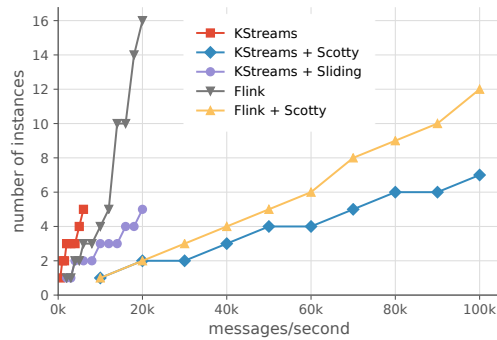
In the following we evaluate the results of the benchmarks. Therefore, we provide resource demand plots. The x-axis of the plots show the number of induced messages/second and the y-axis shows the required number of instances needed to process the load.

In Section 5.3.1, we compare the different frameworks in terms of their scalability. Section 5.3.2, Section 5.3.3, and Section 5.3.4 provide scalability results for Kafka Streams, Flink, and Spark Streaming, respectively.

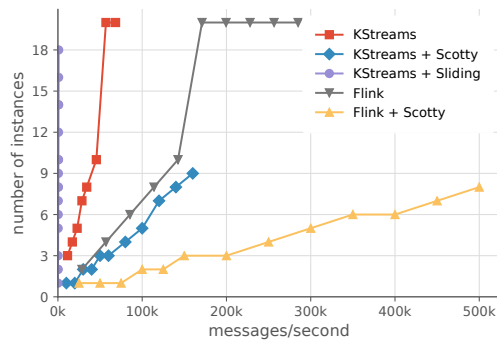
5.3. Results and Discussion



(a) Window size of 30 days and sliding period of 1 day (30 overlapping windows)



(b) Window size of 30 seconds and sliding period of 1 second (30 overlapping windows)



(c) Window size of 5 minutes and sliding period of 1 minute (5 overlapping windows)

Figure 5.1. Theodolite UC3 scalability benchmark results with different window configurations for Kafka Streams with the hopping window, Scotty, and sliding window implementations and for Flink with the hopping window and Scotty implementations.

5.3.1 Comparison of the Frameworks

Theodolite UC3

In Figure 5.1, we provide the results for all the benchmarks of Theodolite's UC3 SUTs. Each plot contains the results for the different SUTs, i.e., for Kafka Streams with the native hopping windows, Scotty and sliding windows, and for Flink with the native hopping windows and Scotty. The plots differ by the defined window configurations of the benchmarks.

In Figure 5.1a, the defined window size is 30 days and the sliding period 1 day and, in Figure 5.1b, the defined window size is 30 seconds and the sliding period is 1 second. Thus, there are 30 overlapping windows in both configurations. In Figure 5.1c, the defined window size is 5 minutes and the sliding period is 1 minute and, as a consequence, there are

5. Experimental Evaluation

5 overlapping windows. In the following, we refer to the different window configurations by their window size, i.e., 30-day window, 30-second window, and 5-minute window.

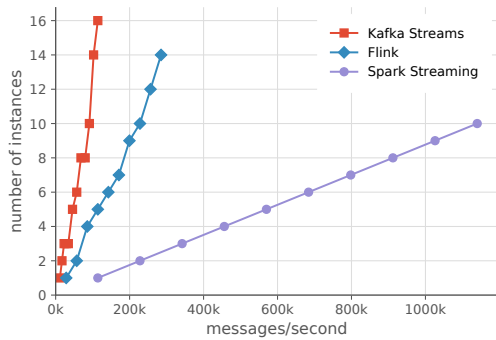
Scotty: The SUTs with Scotty can process the highest load with all window configurations. With the 30-day window and the 5-minute window, the SUT with Flink and Scotty requires fewer instances to process the loads than Kafka streams with Scotty. With the 30-second window, the SUT with Kafka Streams and Scotty requires less instances for processing than the SUT with Flink and Scotty. The Kafka Streams and Flink SUTs with Scotty requires more instances with the 30-second window than with the 30-day and 5-minute windows. Flink with Scotty requires 2 instances to process the load of 100 000 messages/second with the 30-day and 5-minute windows. Though, Flink with Scotty requires 12 instances to process the load of 100 000 messages/second with the 30-second window.

Scotty uses stream slicing for the hopping window aggregation (cf. Section 2.7). Thus, every message is aggregated in one slice and does not need to be aggregated for each window. However, when a window ends, the final result is computed using the partial aggregates of the slices. For the 30-day window with the sliding period of 1 day, final results for a window are not computed in the SLO experiments. The sliding period is 1 day, and the execution time of the experiment is 5 minutes, and, thus, a window does not end during the execution. For the 5-minute window, the sliding period is 1 minute. Therefore, windows end during our experiment and final aggregates are computed. However, the aggregation happens infrequently and does not add an additional resource demand in comparison to the 30-day window. In the 30-seconds window the sliding period is 1 second and, hence, a final aggregate for a window needs to be computed every second. The final aggregation occurs more often and adds an additional resource demand to the computation in comparison to the 30-day and 5-minute windows. Furthermore, Scotty does not provide a fault tolerance mechanism. The partial aggregates are stored in memory, and when an instance fails, the partial aggregates are lost. So, compared to the other SUTs, Scotty has no extra overhead to achieve fault tolerance. The influencing factor of Scotty's resource demand is the sliding period. Shorter sliding periods lead to more final window aggregations and a higher resource demand.

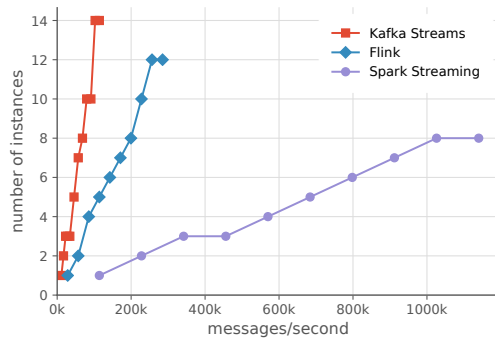
Flink: The Flink SUT with the native hopping window aggregation can process higher loads and requires less resources than the Kafka Streams SUT with the native hopping window aggregation for all window configurations. With the 5-minute window, the resource demand scales linearly up to approximately 140 000 messages/second. To process the next higher load of approximately 170 000 messages/second, the resources are doubled from 10 to 20 instances. Afterwards, the resource demand remains stable up to approximately 300 000 messages/second. We discuss this in more detail in Section 5.3.3.

Kafka Streams: With the 30-day and 5-minute windows, the Kafka Streams SUT with hopping windows can process higher loads than the Kafka Streams SUT with sliding windows. With the 30-second window, it is the opposite. We discuss the sliding window in more detail in Section 5.3.2.

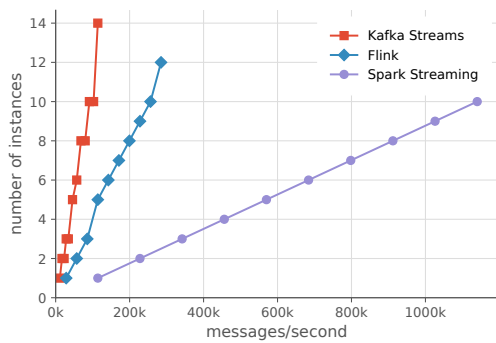
5.3. Results and Discussion



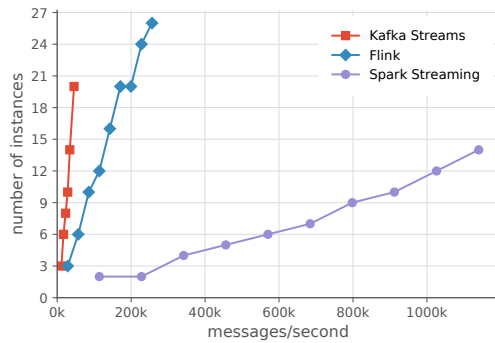
(a) Window size of 150seconds and sliding period of 30seconds (5 overlapping windows)



(b) Window size of 5minutes and sliding period of 1minute (5 overlapping windows)



(c) Window size of 10minutes and sliding period of 2minute (5 overlapping windows)



(d) Window size of 5minutes and sliding period of 15seconds (20 overlapping windows)

Figure 5.2. OSPBench scalability benchmark results for the Kafka Streams, Flink, and Spark Streaming SUTs with different window configurations.

In summary, the SUTs with Scotty can process the highest loads and the sliding period determines the resource demand. For the SUTs with native hopping windows, Flink can process higher loads than Kafka Streams. The resource demand of the SUT with the sliding window depends on the defined window size. With the 30-second window, the sliding window SUT can process higher loads than the Flink and Kafka Streams SUTs with the native hopping window aggregation.

OSPBench

In Figure 5.2, we provide the results for the executions of the OSPBench SUTs. Each plot contains the results of the different SUTs, i.e., for Kafka Streams, Flink, and Spark Streaming with their native window implementations. The plots differ due to the defined window

5. Experimental Evaluation

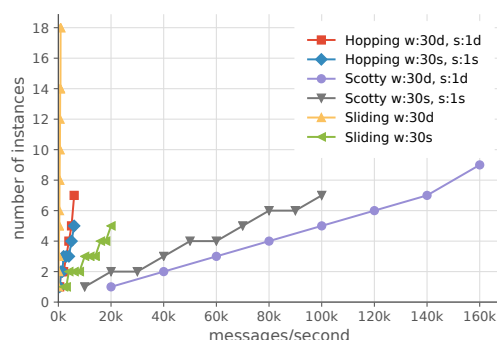


Figure 5.3. Theodolite UC3 scalability benchmark results for Kafka Streams with the hopping window and Scotty implementation with 30 overlapping windows and different window configurations, and sliding window implementations for different time differences (w : window size or time difference, s : sliding period).

configurations in the benchmarks. In Figure 5.2a, Figure 5.2b, and Figure 5.2c the window configurations yield 5 overlapping windows. In Figure 5.2d, the defined window size is 5 minutes and the sliding period 15 seconds and, hence, there are 20 overlapping windows. In the following, we refer to the window configuration with 20 overlapping windows as the 5-minute overlapping window.

The order of the resource demand of the SUTs is the same in all four plots. Kafka Streams has the highest resource demand, followed by Flink, and Spark Streaming has the lowest resource demand. With 5 overlapping windows, the Kafka Streams SUT needs up to 16 instances to process the load that 1 instance of the Spark SUT can process (Figure 5.2a, Figure 5.2b, and Figure 5.2c). With the 5-minute overlapping window, Kafka Streams with up to 30 instances was unable to process the load that 2 instances of Spark can process (Figure 5.2d). The Flink SUT requires with 5 overlapping windows 5 instances to process the load that 1 instance of Spark can process, and 8 to 10 instances to process the load that 2 instances of Spark can process (Figure 5.2a, Figure 5.2b, and Figure 5.2c). With the 5-minute overlapping window, Flink needs 24 instances to process the load that 2 instances of Spark can process (Figure 5.2d).

5.3.2 Kafka Streams Scalability

Figure 5.3 displays the results for the Kafka Streams SUTs with hopping windows, Scotty, and sliding windows with the 30 days and 30 seconds window sizes. In the legend, the window size or time difference is shortened with w and the sliding period is shortened with s . Independently of the window configuration, the Scotty SUT can process the highest loads. The sliding window SUT with the time difference of 30 seconds can process the next highest loads. With the 30 days time difference, the sliding window SUT can process the

5.3. Results and Discussion

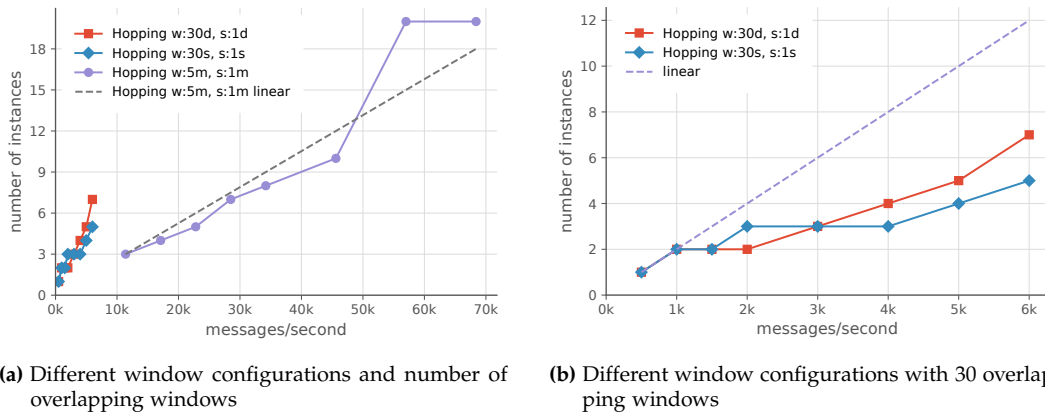


Figure 5.4. Theodolite UC3 scalability benchmark results of the Kafka Streams SUT with hopping windows for different window configurations (w: window size, s: sliding period).

lowest load. The resource demand of the hopping window SUTs is in between both sliding window executions.

Kafka Streams' Hopping Window

In Figure 5.4 and Figure 5.5, we present the results for the scalability of the Kafka Streams hopping window. Figure 5.4 shows the results for the Theodolite UC3 benchmarks and Figure 5.5 the results for the OSPBench benchmarks.

In Figure 5.4a, the benchmarks of the UC3 SUT with the native hopping windows are shown for different window configurations. The window size is denoted with w and the sliding period with s . The dashed line labeled *Hopping w:5m, s:1m linear* is an approximation for a lower bound of linear scalability of the hopping window execution with the window size of 5 minutes and the sliding period of 1 minute. It is created based on the highest load that the lowest number of instances can process. Based on this load, a load per instance is computed and the linear line plotted. The line is used to give a tendency if an execution is linearly scalable.

The SUT can process with the 5-minute window higher loads than with the 30-day and 30-second windows. The 5-minute window has a sliding period of 1 minute and, thus, 5 overlapping windows. The other two window configurations have 30 overlapping windows. Kafka Streams uses buckets for windowed aggregations (cf. Section 2.2.2). In the Theodolite UC3 SUT, partial aggregates are stored in the buckets. Hence, the executions with 30 overlapping windows need to perform more aggregations for each message than the execution with 5 overlapping windows. Therefore, the window configuration with 5 overlapping windows can process higher loads. The SUT scales linearly up to the load of 45 000 messages/second with the 5-minute window and 10 instances are required to

5. Experimental Evaluation

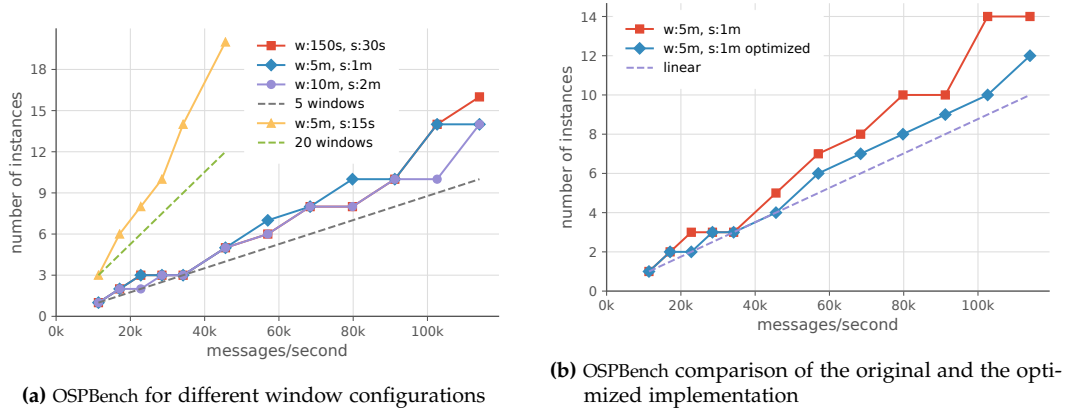


Figure 5.5. OSPBench scalability benchmark results of the Kafka Streams SUT with hopping windows for different window configurations (w: window size, s: sliding period).

process this load. To process the next higher load of 57 000 messages/second, 20 instances are required. However, the load of 68 400 messages/second also requires 20 instances for processing. The resource demand of the SUT is close to the linear approximation with the 5-minute window.

For the 30 overlapping windows, we provide with Figure 5.4b a dedicated plot to evaluate the scalability. The dashed line represents a lower bound for linear scalability. Only one is created since both executions can handle the same load with one instance. Both of the executions are linearly scalable. The executions require up to 3 000 messages/second almost the same number of instances to process the load. Then, the execution with the 30-second window needs 1 to 2 instances less.

Figure 5.5a shows the Kafka Streams OSPBench benchmarks with the different window configurations. We refer to the window configuration with the window size of 5 minutes and a sliding period of 15 seconds as 5-minute overlapping window. There are 20 overlapping windows with the 5-minute overlapping window. With the other window configurations, there are 5 overlapping windows. The dashed lines provide lower bounds for linear scalability. The 20 windows line is for the 5-minute overlapping window and the 5 windows line for the other window configurations.

With the 5-minute overlapping window, the resource demand of the SUT is higher than with the other window configurations for equal loads. The resource demand increases linearly but steeper than the approximation for the linear scalability with the 5-minute overlapping window. With 5 overlapping windows, the resource demand is nearly the same for the executions and scales linearly regardless of the window configuration. It confirms the observations of the UC3 benchmarks. The resource demand increases with more overlapping windows and is independent of the window configuration for the same number of overlapping windows.

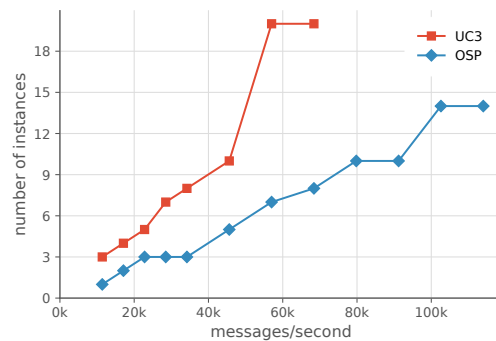


Figure 5.6. Theodolite UC3 and OSPBench comparison of the Kafka Streams SUTs with hopping windows with a window size of 5 minutes and a sliding period of 1 minute (5 overlapping windows).

Figure 5.5b compares the OSPBench Kafka Streams SUTs with the original and the optimized topologies (cf. Figure 4.3). The optimized SUT, which does not perform a repartition, requires one to four instances less for the executed loads. The original implementation perform a repartition when the key is selected. This creates an internal repartition topic into which the data is written. The windowed aggregation processing step reads the data from the repartition topic and more network traffic is produced than in the optimized version.

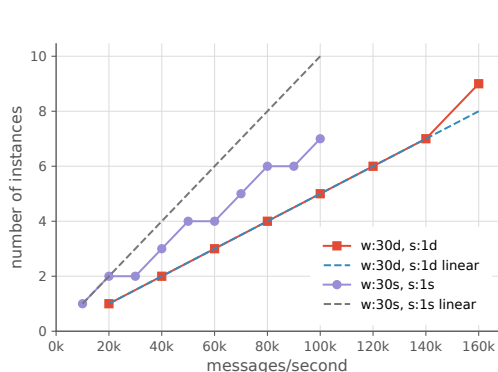
In Figure 5.6 we compare the scalability of the Kafka Streams hopping window implementations of Theodolite and OSPBench. Therefore, we execute both SUTs with the window size of 5 minutes and the sliding period of 1 minute. The UC3 execution needs more resources than the OSPBench execution. Both are scalable, with UC3 requiring more resources and showing a greater increase in resource demand.

Kafka Streams with Scotty

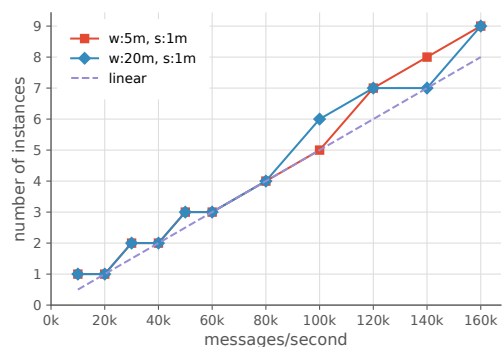
We evaluate the scalability of Scotty with Kafka Streams for different window configurations with Figure 5.7. In Figure 5.7a, we compare two executions with 30 overlapping windows and different window scales. The 30-second window has a higher resource demand than the 30-day window and requires up to 2 more instances for the same load. The SUT has a higher resource demand with the 30-second than with the 30-day window and requires up to 2 more instances for the same load. We discuss the reason for the higher resource demand in Section 5.3.1. Both window configurations scale linearly.

In Figure 5.7b, we compare two executions with different numbers of overlapping windows and the same sliding period. Both executions have a sliding period of 1 minute. There are 5 overlapping windows with the 5-minute window and 20 overlapping windows with the 20-minute window. The resource demand is the same for all but two loads and

5. Experimental Evaluation

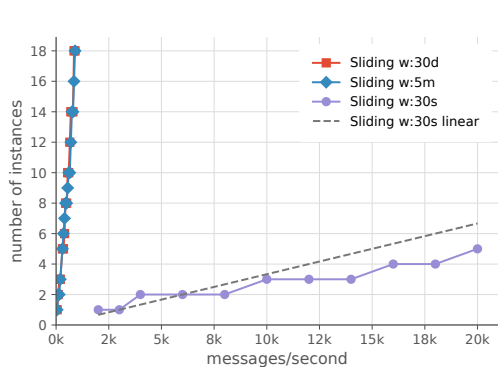


(a) Scotty with 30 overlapping windows

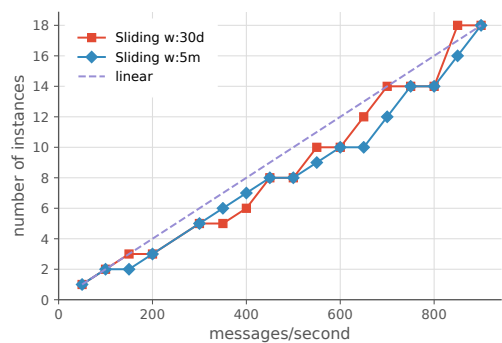


(b) Scotty with different numbers of overlapping windows

Figure 5.7. Theodolite UC3 scalability benchmark results with Kafka Streams and Scotty for different window configurations (w: window size, s: sliding period).



(a) Sliding window with different time differences



(b) Sliding window with 30 days and 5 minutes time difference

Figure 5.8. Theodolite UC3 scalability benchmark results for Kafka streams with sliding windows and different window configurations (w:time difference).

the SUT scales linearly with both window configurations. Thus, the number of overlapping windows does not influence the resource demand and scalability in our benchmarks.

Kafka Streams' Sliding window

In Figure 5.8, we provide the benchmark results for the sliding window SUT. With the 30-second window, the SUT has the lowest resource demand and scales linearly (Figure 5.8a). Moreover, the 5-minute and 30-day window executions cannot process the load that 1 instance can process with the 30-second window. The sliding window is data driven and,

thus, the number of overlapping windows depends on the data (cf. Section 2.4). Since the execution time for the experiments is 5 minutes, new windows are created during the whole execution for the 30-day and the 5-minute window configurations. Thus, the number of overlapping windows increases constantly and, in the end, there are 300 overlapping windows. For the 30-second window, Kafka Streams with sliding windows creates 30 overlapping windows in the execution.

Figure 5.8b provides a higher resolution of the 5-minute and 30-day window executions. Both of them have a similar resource demand and scale linearly in our benchmark.

Kafka Streams Summary

In summary, the SUT with Scotty can process the highest loads. Scotty's resource demand is influenced by the defined sliding period (Figure 5.7a). With shorter sliding periods, the resource demand increases. However, the differences in resource demand for longer and shorter sliding periods are at most 2 instances in our benchmarks.

Kafka Streams' hopping window aggregation scales linearly and the resource demand depends on the number of overlapping windows (Figure 5.5). The more overlapping windows exist, the more instances are required for processing. For the same amount of overlapping windows, the resource demand is independent of the window size and sliding period. In Kafka Streams, the resource demand can be reduced by avoiding unnecessary repartition of the data.

Kafka Streams' sliding window aggregation scales linearly. The resource demand of the sliding window depends on the aggregation data since the data determines the number of overlapping windows (Figure 5.8). The higher the number of overlapping windows is, the higher the resource demand is. When the data rate is known, the time difference can be used to limit the number of overlapping windows. In our experiments, the sliding window aggregation has a lower resource demand than the hopping window aggregation when the number of overlapping windows is the same.

5.3.3 Flink Scalability

Figure 5.9 displays the results for the Flink implementations with hopping windows and Scotty for the 30 days and 30 seconds window sizes. In the legend the window size is shortened with w and the sliding period is shortened with s .

The Scotty SUT can process the highest loads. With the 30-second window, the Scotty SUT needs many more instances than with the 30-day window. The load that 1 or 2 instances can process using the 30-day window requires 5 or 12 instances with the 30-second window. In Flink, the hopping window SUT have the higher resource. The hopping window SUT can process higher loads with the 30-day window than with the 30-second window.

5. Experimental Evaluation

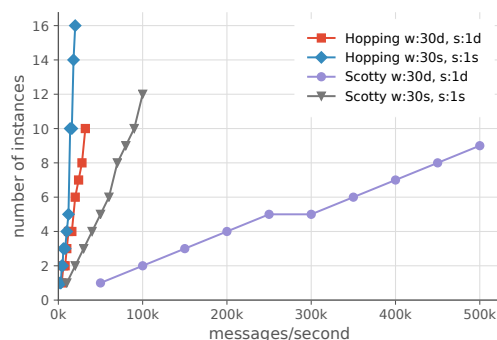
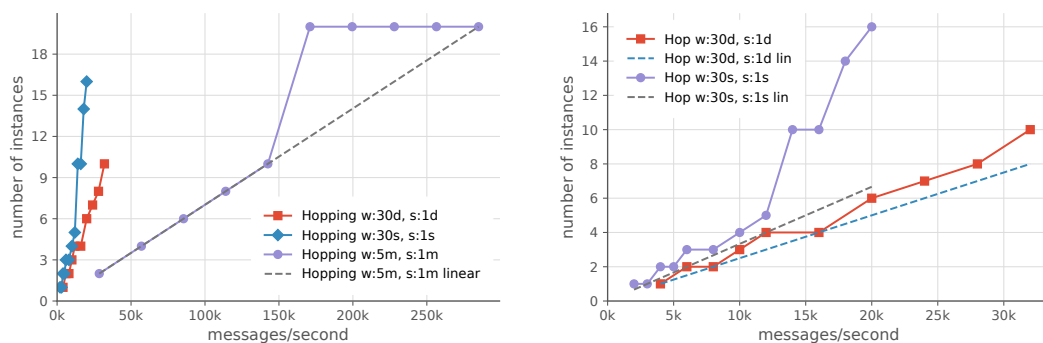


Figure 5.9. Theodolite UC3 scalability benchmark results for Flink with the hopping window and Scotty implementations with 30 overlapping windows and different window configurations (w : window size, s : sliding period).



(a) Theodolite UC3 with hopping windows and different window sizes

(b) Theodolite UC3 for 30 overlapping windows and different window configurations

Figure 5.10. Theodolite scalability benchmark results for Flink with hopping windows and Scotty with different window configurations (w : window size, s : sliding period).

Flink's Hopping Windows

Figure 5.10 shows the results for the Flink UC3 SUT with hopping windows. The window size is denoted with w and the sliding period with s . The dashed lines are linear approximations for the lower bound linear scalabilities.

Figure 5.10a shows that the SUT can process higher loads with the 5-minute window than with the 30-day and 30-second windows. The 5-minute window has a sliding period of 1 minute and, thus, 5 overlapping windows. The other two window configurations have 30 overlapping windows. Like Kafka Streams, Flink uses buckets for windowed aggregations (cf. Section 2.2.2). Thus, the executions with 30 overlapping windows need to perform more aggregations for each message than the execution with 5 overlapping

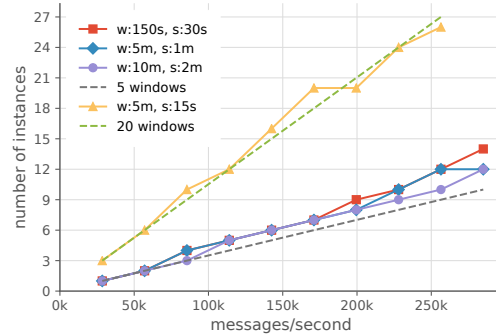


Figure 5.11. OSPBench scalability benchmark results for Flink with hopping windows and different window configurations (w: window size, s: sliding period).

windows. Therefore, the window configuration with 5 overlapping windows can process higher loads. The SUT scales linearly up to the load of 140 000 messages/second with the 5-minute window and 10 instances are required to process this load. For processing the next higher load of 170 000 messages/second, 20 instances are required. However, loads up to 285 000 messages/second also require 20 instances for processing. Henning and Hasselbring [2021a] observed the effect that the performance is better if the number of partitions is a multiple of the number of instances. In our experiment we have 40 partitions, and, hence, a multiple of 20 instances. The resource demand with the 5-minute window is for the highest load like the approximation and, thus, we conclude linear scalability.

Figure 5.10b shows the benchmark results in more detail with the 30 overlapping windows. For loads up to 12 000 messages/second, both of the window configurations have nearly the same resource demand. With the 30-second window 1 instance more is required than with the 30-day window. From 12 000 messages/second to 14 000 messages/second the resource demand is doubled from 5 to 10 instances with the 30-second window. The 30-second window execution scales steeper than linear. As opposed to this, the 30-day window execution scales linearly.

Figure 5.11 shows the benchmark results of the OSPBench SUT with different window configurations. The window configuration with a window size of 5 minutes and a sliding period of 15 seconds is referred to as 5-minute overlapping window. We refer to the other window configurations by their window size. There are 20 overlapping windows for the 5-minute overlapping window and 5 overlapping windows for the other window configurations. The dashed lines provide lower bounds for linear scalability. The 20 windows line is for the 5-minute overlapping window and the 5 windows line for the other window configurations.

The SUT has a higher resource demand with the 5-minute overlapping window than with the other window configurations for processing the same loads. With the 5-minute overlapping window, the resource demand increases linearly and is like the linear approxi-

5. Experimental Evaluation

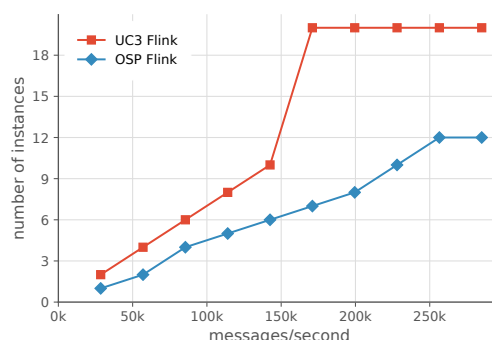


Figure 5.12. Theodolite UC3 and OSPBench comparison of the Flink SUTs with hopping windows with a window size of 5 minutes and a sliding period of 1 minute (5 overlapping windows).

mation. The resource demand with the 5 overlapping window configurations differ in a maximum of 2 instances and the SUT scales linearly. The OSPBench results do not confirm the observations of the UC3 benchmarks that the resource demand scales steeper than linear (Figure 5.10b) with more overlapping windows.

In Figure 5.12 we compare the scalability of the Flink hopping window SUTs of Theodolite and OSPBench. Therefore, we execute both SUTs with the window size of 5 minutes and the sliding period of 1 minute. Other configurations are not further adjusted. The UC3 execution needs more resources than the OSPBench execution. Both are scalable, but the resource demand curve is steeper for UC3.

Flink with Scotty

Figure 5.13 evaluates the scalability of Scotty with Flink for different window configurations. In Figure 5.13a we compare two executions with 30 overlapping windows and different window scales. The SUT has a higher resource demand with the 30-second window than with the 30-day window. 12 instances are required with the 30-second window to process the load that 2 instances can process with the 30-day window. We discuss the reason for this in Section 5.3.1. Both executions scales linearly.

In Figure 5.13b we compare two executions with different numbers of overlapping windows and the same sliding period. The sliding period is in both executions 1 minute. There are 5 overlapping windows with the 5-minute window and 20 overlapping windows with the 20-minute window. The resource demand is the same for all but two loads and the SUT scales linearly with both window configurations. Thus, the number of overlapping windows has in our benchmarks no influence on the resource demand and scalability.

5.3. Results and Discussion

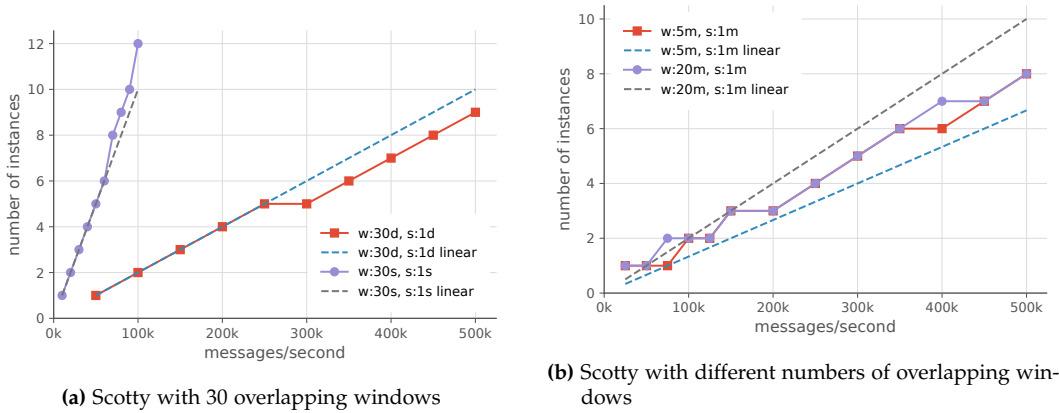


Figure 5.13. Theodolite UC3 with Flink and Scotty for different window configurations (w: window size, s: sliding period).

Flink Summary

The SUT with Flink and Scotty can process the highest loads. The resource demand of Scotty is influenced by the defined sliding period (Figure 5.13a). With shorter sliding periods, the resource demand increases. In contrast to Kafka Streams, the shorter sliding period has a higher influence on the resource demand of Flink with Scotty. In our experiments, the resource demand is up to 6 times higher with the shorter sliding period than with the longer sliding period.

The resource demand of the Flink hopping window aggregation depends on the number of overlapping windows (Figure 5.11). The more overlapping windows exist, the more instances are required for processing. In UC3, the resource demand for the same number of overlapping windows depends on the scaling of the window size and the sliding period. However, scaling windows with the same number of overlapping windows does not affect the resource demand in OSPBench. The resource demand of the SUTs with Flink’s hopping window is smaller than the SUTs with Kafka Streams’ hopping window.

5.3.4 Spark Streaming Scalability

Figure 5.14 shows the benchmarks of the different window configurations in OSPBench with Spark Streaming. The dashed line provides a lower bound for linear scalability for all the executions. We refer to the window configuration with the window size of 5 minutes and a sliding period of 15 seconds as 5-minute overlapping window and to the others by their window size. There are 20 overlapping windows for the 5-minute overlapping window and 5 overlapping windows for the other window configurations.

5. Experimental Evaluation

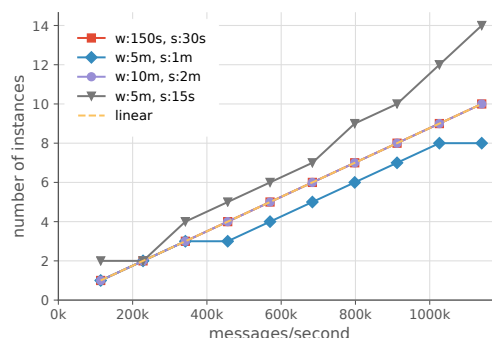


Figure 5.14. OSPBench scalability benchmark results for Spark Streaming with hopping windows and different window configurations (w: window size, s: sliding period).

The SUT has a higher resource demand with the 5-minute overlapping window and requires 1 to 6 instances more than with the other window configurations to process the same loads. The SUT has the lowest resource demand with the 5-minute window and requires 1 instance less than with the 150-second and 10-minute windows. In all the benchmarks the resource demand increases linearly and is like the linear approximation for scalability.

Out of the benchmarked frameworks, Spark Streaming can process the highest loads (Figure 5.2). For the same number of overlapping windows, the window size and sliding period do not influence the resource demand. Furthermore, with more overlapping windows, the required amount of instances is less than a factor of 2 higher to process the same loads.

5.4 Threats to Validity

In the following we present the threats to the validity of our evaluation. We distinguish between internal validity (Section 5.4.1) and external validity (Section 5.4.2).

5.4.1 Internal Validity

Lag Trend Ratio

The *lag trend ratio* for the executions is set to 5 percent and to 5, 8, or 15 percent for Spark. This ratios can be too low or too high. Especially for Spark, the computed lag trend depends on the evaluated period of the execution. Thus, the ratio for Spark must take this into account. However, we do not attempt to specify the exact number of instances required to process a load, but rather show scalability. If each execution requires a few more or fewer instances, the scalability results are not affected.

Tested loads and instances

The tested loads and instances determine the possible resource demand functions. If they are not correctly chosen they can highly influence the results. If they are too coarse, the accuracy of the results will decrease.

Execution Time

The execution time for UC3 is set to 5 minutes and for OSPBench to 6 or 7 minutes. Since windowed aggregations are stateful operations, a state needs to be maintained. For longer executions the state might increase. However, the state size is fixed for each window. The aggregations are incremental and the result is a single object. Changes of the objects for the partial aggregation are only values of numerical data types. All aggregations, except for the sliding window aggregation, have the same number of parallel windows at any time during the execution. When the SLO experiment execution is longer than the sliding period of the window, old states exist and need to be removed. For executions where the sliding period of the window is larger than the execution time, no old states exist. Thus, removing old states may influence the results of executions where the sliding period is longer than the execution time.

Sliding windows create new windows triggered by new data. Thus, the number of overlapping windows is determined by the data. When the SLO experiment execution is longer than the defined time difference, the maximum number of parallel windows are created. If the time difference is greater than the execution time, the maximum number of parallel windows are not reached. Thus, for those windows the results may change for longer execution times. For example, if the execution time is 30 days, the 30-day window would create $30 * 24 * 60 * 60 = 2\,592\,000$ overlapping windows in our benchmark.

Warmup Time

At the start of the experiments the SUTs need some time to set up and may not start directly the processing of data generated by the load generator. Thus, at the beginning there is already a record lag before processing is started. Therefore, the SUTs get some time to start and process the queued messages. A warmup of 1 minute is defined to catch up with the processing.

Repetitions

We repeat the experiments three times to counter influences like the pod assignment in Kubernetes. From the three executions the median record lag is chosen and is used to counter outliers. To increase the validity of the results the experiments should be executed with more repetitions.

5. Experimental Evaluation

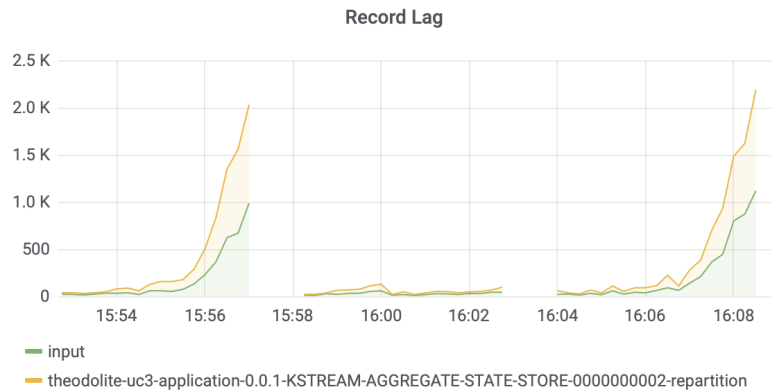


Figure 5.15. Change of the hour of day during the execution of the UC3 sliding window SUT with a time difference of 30 days.

UC3 Change of the Hour of Day Value

In the task sample UC3 the data is aggregated based on the temporal attribute hour of day. For example, for the times from 3h to 4h, the hour of day is 3, and from 4h to 5h, the hour of day is 4. If the hour of day value for the ingested data changes, also the selected keys change. The key is composed of the hour of the day and the original key.

Especially for the sliding window aggregation, this has a huge impact. Figure 5.15 shows the record lags of three repetitions from one SLO experiment. The SLO experiments on the left and the right have a steep increase in the record lag. The record lag of the SLO experiment in the middle increases slowly until 16h, then drops and increases again slowly until the end of the experiment. With sliding windows, a new window is created with each record and the record must be aggregated in all previous windows to which it belongs. In our experiments, the number of overlapping windows increases every second until the execution is as long as the defined time difference. The more overlapping windows there are, the more time is needed for processing. When the hour of day value changes, data with new keys arrive and new aggregations are started with no overlapping windows. Thus, fewer aggregations need to be performed for the new data. However, we perform 3 repetitions of the SLO experiment and take the median threshold. When the window size is below 1 hour, this can only happen in one of the three repetitions at most and does not influence the result since the median record lag is chosen.

Kafka Streams Suppress

The UC3 Kafka Streams sliding window SUT and the Kafka Streams OSPBench SUT use the suppress operator in their executions. Normally Kafka Streams continuously updates new results in Kafka [Apache Software Foundation 2021c]. The suppress can delay updates and

5.4. Threats to Validity

limit the rate [Wang et al. 2021; Apache Software Foundation 2021c]. In the benchmarked applications only final window results are published to Kafka. Therefore, suppress change the logic of the application.

In the UC3 SUT with the hopping window aggregation, suppress is not used. Thus, a comparison of the Kafka Streams SUTs with and without suppress may not be meaningful. However, the sliding window aggregation has different semantics than the hopping window aggregation and outputs for every record a final window result. Moreover, we do not derive general results from the comparison of the UC3 and OSPBench SUTs.

5.4.2 External Validity

Execution infrastructure

The experiments are only executed in our Kubernetes Cluster. Other clusters have different configurations of CPU, Memory, and bandwidth. Thus, the results may not apply to them. Furthermore, the different frameworks may be sensible to different limiting factors. Therefore, the experiments should be repeated on other Kubernetes clusters.

Aggregation Function

OSPBench uses a distributive aggregation and benchmark UC3 an algebraic aggregation. No holistic aggregation is performed. Therefore, the results cannot be generalized to holistic window aggregations.

Related Work

6.1 Scalability Benchmarking of SPEs

In different studies, the scalability of SPEs are evaluated. Karakaya et al. [2017] perform a comparison of the SPEs Apache Flink, Apache Spark, and Apache Storm. The authors use the Yahoo Streaming Benchmarks and optimize the SPEs to their ideal performance. As part of their evaluation they define a scale-up ratio for the comparison of scalability. The scale-up ratio compares how many events are processed with at least two worker instances compared to only one worker instance.

Henning and Hasselbring [2021d] use Theodolite to benchmark the scalability of Kafka Streams and Flink for different deployment options. Further, the impact of these different deployment options on the scalability is evaluated. The deployment options are applied to the benchmark applications UC1, UC2, UC3, and UC4. The experiments with UC3 are executed with a window size of 3 days and a sliding period of 1 day, i.e., having 3 overlapping windows. The number of partitions on Kafka, the commit interval of Kafka Streams, the checkpointing of Flink, and CPU and Memory per Kubernetes pod are evaluated with different configurations. The results are that Kafka Streams scales independently of the number of Kafka partitions and provided Kubernetes resources. Kafka streams require more instances with shorter commit intervals. However, Kafka Streams scales independently of the tested commit intervals. Flink scales linearly without checkpointing and with a checkpoint interval of 10 seconds, but the resource demand is higher with checkpointing. Finally, Kafka Streams and Flink are compared and the authors conclude that the scalability of both are quite similar. In their UC3 comparison Flink requires constantly slightly more instances than Kafka Streams.

Van Dongen and Van den Poel [2021b] extend OSPBench to perform scalability benchmarks. The authors identify factors that influence the scalability in distributed stream processing. Therefore, they determine the throughput bottlenecks and their influence on scalability. An enrichment pipeline and an aggregation pipeline are benchmarked. The enrichment pipeline ingest two data streams, parse the data into Scala classes, joins them with one-second intervals, and aggregates the joined streams with tumbling window of 1 second. In the aggregation pipeline one stream is ingested, the data is parsed and a hopping window aggregation with a window size of 5 minutes and a sliding period of 1 minute is performed. The aggregation in this pipeline is performed non-incrementally.

6. Related Work

Apache Flink, Kafka Streams, Spark Streaming, and Structured Streaming are used to implement the pipelines. In their benchmarks the authors determine the peak sustainable throughput. The peak sustainable throughput is the maximum throughput under which the processing framework is stable. A framework must meet three conditions to be stable. (1) the input queues and the latency of the output are not continuously increasing. Further, (2) the CPU utilization is below 80% and (3) the median latency is below a defined threshold. The authors state that the throughput increase is the main objective of scaling and define a scaling efficiency metric. The metric is based on the peak sustainable throughput and utilize two auxiliary functions. The throughput increase factor (TIF) gives information about the increase of throughput as the result of scaling and the resource increase factor (RIF) provides information on how many additional resources are provided. Finally, the scaling efficiency is computed by dividing TIF with RIF. Moreover, they describe the bottlenecks of the frameworks for the different pipelines and list five factors that influence the scalability.

6.2 Optimized Hopping Window Implementations

Different optimizations for the hopping window aggregation in distributed stream processing are proposed. Traub et al. [2021] present Scotty and evaluate the performance of stream slicing compared to other techniques and built-in implementations of SPEs. They use real-world sensor data and measure the throughput, latency, and memory consumption in their experiments. To show the scalability of Scotty the authors change the number of cores for processing. For comparison, the same is done with a Flink implementation. By using the throughput they deduce that Scotty and Flink scales linear with the number of cores in their application and Scotty achieves a magnitude higher throughput than Flink. Moreover, they compare the throughput of Scotty with built-in solutions of Apache Spark, Apache Samza, Kafka Streams, Apache Flink, Apache Storm, and Apache Beam. The authors evaluate in the experiment how the throughput scale for different number of parallel windows.

Gomes et al. [2021] propose with Railgun a new distributed stream processing engine. It provides real time sliding window aggregations under their defined MAD requirements. The MAD requirements are that the latencies are at high percentiles in the millisecond level, sliding window aggregations are accurate, and the processing is distributed, scalable, and fault tolerant. A SQL like language is used and it provides sliding windows, tumbling windows, and infinite windows. Further, the different aggregation functions are predefined. The authors perform evaluations on Railgun comparing it to Flink, scaling the windows, and scaling the nodes. They state that Railgun scales nearly linearly.

Zhang et al. [2021] present the Parallel Boundary Aggregator (PBA), a new algorithm for efficient hopping window aggregations. It utilizes slicing and two buffers to compute the aggregation. The authors implement PBA on top of Apache Flink. They perform

6.2. Optimized Hopping Window Implementations

experiments with different number of resources and compare the throughput of Flink to Flink with PBA. The throughput of Flink with PBA is higher and the authors observe that the throughput of both solutions scale linear with the resources.

Conclusions and Future Work

7.1 Conclusions

In this thesis, we empirically evaluate the scalability of hopping window aggregations with the Theodolite benchmarking method. We apply two different benchmark applications. One is Theodolite UC3, an application benchmark, and one is the OSPBench aggregation pipeline, a microbenchmark. Both benchmark applications are implemented with different frameworks and windowing methods. Furthermore, we execute the benchmarks with different window configurations, varying the number of overlapping windows, the window size, and the sliding period.

Independent of the window configuration, Spark Streaming has the lowest resource demand, followed by Flink, and Kafka Streams has the highest resource demand in our experiments. Further, we benchmark Scotty in combination with Kafka Streams and Flink. In both SPEs, Scotty can process higher loads than the native windowing implementations. However, Scotty does not provide any fault tolerant mechanisms. Furthermore, we implement the new introduced sliding window of Kafka Streams into UC3 and benchmark it. For a short execution time and having the same amount of overlapping windows it outperforms the Kafka Streams and Flink hopping window aggregations.

In all implementations, except for Scotty, the resource demand increases with more overlapping windows. The number of overlapping windows has a smaller impact on Spark Streaming's resource demand than for Kafka Streams and Flink. For Scotty, the resource demand increases with shorter sliding periods.

Moreover, it is the first time a benchmark application not contained within Theodolite is benchmarked with Theodolite. We demonstrate the extensibility of Theodolite by integrating OSPBench.

7.2 Future Work

Scotty uses a memory state store in the connector implementations. In case of failures the state is lost. If another instance takes over the task, processing starts with the last offset. However, the old state is lost and data is missing in the computations. In contrast, the native SPE implementations use mechanisms to persist the state. Thus, they are more fault tolerant at the cost of additional overhead. Therefore, the native solutions of the SPEs for

7. Conclusions and Future Work

storing states should be implemented into the Scotty connector. Then, Scotty should be benchmarked and evaluated again.

Besides Scotty, the Parallel Boundary Aggregator [Zhang et al. 2021] also uses a slicing method to provide efficient computations of hopping windows. They implemented their solution in Flink. Thus, it would be interesting to use the implementation in either the Theodolite or OSPBench SUTs, benchmark the scalability and compare PBA to the other implementations.

OSPBench [van Dongen 2021] provides 9 pipelines in total. The pipelines are stateless and stateful and vary in complexity. We benchmarked the scalability of one aggregation pipeline. Thus, there are 8 more pipelines where the scalability can be benchmarked.

In the OSPBench aggregation pipeline the native hopping window implementations are used. To confirm the benchmark results for the SUTs with the sliding window and Scotty, these alternative windowing methods can also be used in the aggregation pipeline.

We use the *lag trend ratio* SLO to determine the resource demand of the Spark Streaming SUT. The commit of offsets in Spark Streaming SUT depends on the batch interval and the sliding period and, thus, the record lag fluctuates between highs and lows. Further, Structured Streaming does not commit the offsets at all which prevent us to benchmark its scalability. However, Spark provides metrics that can be used as an SLO to indicate that the messages are not queuing up. For example, Spark provides metrics for the processing time of a job. If the processing time of a job is below the window time the application can keep up processing the data. Therefore, new SLOs for Spark Streaming and Structured Streaming can be implemented and both frameworks benchmarked.

Bibliography

- [Akidau et al. 2015] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, out-of-Order Data Processing. *Proc. VLDB Endow.* 8.12 (Aug. 2015), pages 1792–1803. DOI: 10.14778/2824032.2824076. (Cited on pages 1, 5–8, 10)
- [Apache Software Foundation 2021a] Apache Software Foundation. *Apache Beam Documentation*. Accessed on December 9, 2021. 2021. URL: <https://beam.apache.org/documentation/>. (Cited on page 6)
- [Apache Software Foundation 2021b] Apache Software Foundation. *Apache Flink Documentation*. Accessed on December 9, 2021. 2021. URL: <https://nightlies.apache.org/flink/flink-docs-release-1.14/>. (Cited on pages 6, 13–15, 43)
- [Apache Software Foundation 2021c] Apache Software Foundation. *Apache Kafka Documentation*. Accessed on December 9, 2021. 2021. URL: <https://kafka.apache.org/documentation/>. (Cited on pages 2, 6–9, 11, 13, 39, 42, 43, 70, 71)
- [Apache Software Foundation 2021d] Apache Software Foundation. *Apache Spark Documentation*. Accessed on December 9, 2021. 2021. URL: <https://spark.apache.org/docs/latest/index.html>. (Cited on pages 15, 16, 44–46)
- [Apache Software Foundation 2021e] Apache Software Foundation. *Apache Wiki*. Accessed on December 9, 2021. 2021. URL: <https://cwiki.apache.org>. (Cited on pages 8, 9)
- [Arasu et al. 2004] A. Arasu, A. S. Maskey, M. Cherniack, E. Ryvkina, E. Galvez, M. Stonebraker, D. Maier, and R. Tibbetts. Linear Road: A Stream Data Management Benchmark. In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*. VLDB '04. Toronto, Canada: VLDB Endowment, Aug. 2004, pages 480–491. DOI: <https://doi.org/10.1016/B978-012088469-8.50044-9>. (Cited on pages 31–33)
- [Armbrust et al. 2018] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, May 2018, pages 601–613. DOI: 10.1145/3183713.3190664. (Cited on page 16)
- [Beyer et al. 2016] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Incorporated, Apr. 2016. 524 pages. (Cited on page 19)

Bibliography

- [Bordin et al. 2020] M. V. Bordin, D. Griebler, G. Mencagli, C. F. R. Geyer, and L. G. L. Fernandes. DSPBench: A Suite of Benchmark Applications for Distributed Data Stream Processing Systems. *IEEE Access* 8 (Dec. 2020), pages 222900–222917. DOI: 10.1109/ACCESS.2020.3043948. (Cited on pages 2, 32, 33)
- [Carbone et al. 2015] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin* 38 (Jan. 2015). (Cited on pages 13–15)
- [Carbone et al. 2018] P. Carbone, A. Katsifodimos, and S. Haridi. Stream Window Aggregation Semantics and Optimization. In: *Encyclopedia of Big Data Technologies*. Edited by S. Sakr and A. Zomaya. Cham: Springer International Publishing, 2018, pages 1–9. DOI: 10.1007/978-3-319-63962-8_154-1. (Cited on page 1)
- [Cherniack et al. 2003] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In: *Conference on Innovative Data Systems Research*. 2003. (Cited on page 5)
- [Chintapalli et al. 2016] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2016, pages 1789–1792. DOI: 10.1109/IPDPSW.2016.138. (Cited on pages 31, 33, 35)
- [Cloud Native Computing Foundation 2021] Cloud Native Computing Foundation. *Kubernetes Documentation*. Accessed on December 10, 2021. 2021. URL: <https://kubernetes.io/docs/home/>. (Cited on page 20)
- [Gomes et al. 2021] A. S. Gomes, J. Oliveirinha, P. Cardoso, and P. Bizarro. Railgun: Managing Large Streaming Windows under MAD Requirements. *Proc. VLDB Endow.* 14.12 (July 2021), pages 3069–3082. DOI: 10.14778/3476311.3476384. (Cited on pages 8, 74)
- [Hasselbring 2021] W. Hasselbring. Benchmarking as Empirical Standard in Software Engineering Research. In: *Evaluation and Assessment in Software Engineering*. EASE 2021. Trondheim, Norway: Association for Computing Machinery, June 2021, pages 365–372. DOI: 10.1145/3463274.3463361. (Cited on page 2)
- [Henning and Hasselbring 2020] S. Henning and W. Hasselbring. Toward Efficient Scalability Benchmarking of Event-Driven Microservice Architectures at Large Scale. In: *11th Symposium on Software Performance*. Volume 40. Softwaretechnik-Trends 3. Nov. 2020, pages 28–30. (Cited on pages 18, 21, 53)
- [Henning and Hasselbring 2021a] S. Henning and W. Hasselbring. A Configurable Method for Benchmarking Scalability of Cloud-Native Applications (2021). Under Review. (Cited on pages 21, 53, 65)

- [Henning and Hasselbring 2021b] S. Henning and W. Hasselbring. How to Measure Scalability of Distributed Stream Processing Engines? In: *Companion of the ACM/SPEC International Conference on Performance Engineering*. ICPE '21. Virtual Event, France: Association for Computing Machinery, Apr. 2021, pages 85–88. DOI: 10.1145/3447545.3451190. (Cited on pages 18, 19, 21)
- [Henning and Hasselbring 2021c] S. Henning and W. Hasselbring. The Titan Control Center for Industrial DevOps analytics research. *Software Impacts* 7 (Feb. 2021). DOI: <https://doi.org/10.1016/j.simpa.2020.100050>. (Cited on pages 1, 4, 18)
- [Henning and Hasselbring 2021d] S. Henning and W. Hasselbring. Theodolite: Scalability Benchmarking of Distributed Stream Processing Engines in Microservice Architectures. *Big Data Research* 25 (July 2021). DOI: <https://doi.org/10.1016/j.bdr.2021.100209>. (Cited on pages 2, 3, 18–21, 23, 32, 33, 54, 73)
- [Henning et al. 2021a] S. Henning, W. Hasselbring, H. Burmester, A. Möbius, and M. Wojcieszak. Goals and measures for analyzing power consumption data in manufacturing enterprises. 3 (Mar. 2021), pages 65–82. DOI: 10.1007/s42488-021-00043-5. (Cited on pages 1, 21)
- [Henning et al. 2021b] S. Henning, B. Wetzel, and W. Hasselbring. Reproducible Benchmarking of Cloud-Native Applications with the Kubernetes Operator Pattern. In: *Symposium on Software Performance*. Nov. 2021. (Cited on page 20)
- [Hesse et al. 2021] G. Hesse, C. Matthies, M. Perscheid, M. Uflacker, and H. Plattner. ESP-Bench: The Enterprise Stream Processing Benchmark. In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ICPE '21. Virtual Event, France: Association for Computing Machinery, Apr. 2021, pages 201–212. DOI: 10.1145/3427921.3450242. (Cited on pages 2, 32, 33)
- [Jafarpour and Desai 2019] H. Jafarpour and R. Desai. KSQL: Streaming SQL Engine for Apache Kafka. In: *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. Edited by M. Herschel, H. Galhardas, B. Reinwald, I. Fundulaki, C. Binnig, and Z. Kaoudi. OpenProceedings.org, 2019, pages 524–533. DOI: 10.5441/002/edbt.2019.48. (Cited on pages 6–8, 11–13)
- [Karakaya et al. 2017] Z. Karakaya, A. Yazici, and M. Alayyoub. A Comparison of Stream Processing Frameworks. In: *2017 International Conference on Computer and Applications (ICCA)*. 2017, pages 1–12. DOI: 10.1109/COMAPP.2017.8079733. (Cited on pages 31–33, 35, 73)
- [Karimov et al. 2018] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. Benchmarking Distributed Stream Data Processing Systems. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pages 1507–1518. DOI: 10.1109/ICDE.2018.00169. (Cited on pages 31–33, 35)
- [Kreps 2011] J. Kreps. Kafka : A Distributed Messaging System for Log Processing. In: *NetDB Workshop*. 2011. (Cited on pages 11, 12)

Bibliography

- [Laaber and Leitner 2018] C. Laaber and P. Leitner. An Evaluation of Open-Source Software Microbenchmark Suites for Continuous Performance Assessment. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR '18. Gothenburg, Sweden: Association for Computing Machinery, May 2018, pages 119–130. DOI: 10.1145/3196398.3196407. (Cited on page 18)
- [Li et al. 2015] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. SparkBench: A Comprehensive Benchmarking Suite for in Memory Data Analytic Platform Spark. In: *Proceedings of the 12th ACM International Conference on Computing Frontiers*. CF '15. Ischia, Italy: Association for Computing Machinery, May 2015. DOI: 10.1145/2742854.2747283. (Cited on pages 15, 31, 33)
- [Lu et al. 2014] R. Lu, G. Wu, B. Xie, and J. Hu. Stream Bench: Towards Benchmarking Modern Distributed Stream Computing Frameworks. In: *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. 2014, pages 69–78. DOI: 10.1109/UCC.2014.15. (Cited on pages 31–33, 35)
- [Noghabi et al. 2017] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: Stateful Scalable Stream Processing at LinkedIn. *VLDB Endowment* 10.12 (Aug. 2017), pages 1634–1645. DOI: 10.14778/3137765.3137770. (Cited on pages 1, 2)
- [Poggi 2019] N. Poggi. Microbenchmark. In: *Encyclopedia of Big Data Technologies*. Edited by S. Sakr and A. Y. Zomaya. Cham: Springer International Publishing, 2019, pages 1143–1152. DOI: 10.1007/978-3-319-77525-8_111. (Cited on page 18)
- [Ralph 2021] P. Ralph. ACM SIGSOFT Empirical Standards Released. *SIGSOFT Softw. Eng. Notes* 46.1 (Jan. 2021), page 19. DOI: 10.1145/3437479.3437483. URL: <https://doi.org/10.1145/3437479.3437483>. (Cited on page 32)
- [Ralph et al. 2021] P. Ralph, N. bin Ali, S. Baltés, D. Bianculli, J. Diaz, Y. Dittrich, N. Ernst, M. Felderer, R. Feldt, A. Filieri, B. B. N. de França, C. A. Furia, G. Gay, N. Gold, D. Graziotin, P. He, R. Hoda, N. Juristo, B. Kitchenham, V. Lenarduzzi, J. Martínez, J. Melegati, D. Mendez, T. Menzies, J. Molleri, D. Pfahl, R. Robbes, D. Russo, N. Saarimäki, F. Sarro, D. Taibi, J. Siegmund, D. Spinellis, M. Staron, K. Stol, M.-A. Storey, D. Taibi, D. Tamburri, M. Torchiano, C. Treude, B. Turhan, X. Wang, and S. Vegas. *Empirical Standards for Software Engineering Research*. 2021. (Cited on page 2)
- [Sax et al. 2018] M. J. Sax, G. Wang, M. Weidlich, and J.-C. Freytag. Streams and Tables: Two Sides of the Same Coin. In: *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*. BIRTE '18. Rio de Janeiro, Brazil: Association for Computing Machinery, 2018. DOI: 10.1145/3242153.3242155. (Cited on pages 1, 5, 11, 12)
- [Shahverdi et al. 2019] E. Shahverdi, A. Awad, and S. Sakr. Big Stream Processing Systems: An Experimental Evaluation. In: *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. 2019, pages 53–60. DOI: 10.1109/ICDEW.2019.00-35. (Cited on pages 32, 33)

- [Shukla et al. 2017] A. Shukla, S. Chaturvedi, and Y. Simmhan. RIoTBench: An IoT benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience* 29.21 (Oct. 2017). DOI: <https://doi.org/10.1002/cpe.4257>. (Cited on pages 31, 33)
- [Sim et al. 2003] S. E. Sim, S. Easterbrook, and R. C. Holt. Using Benchmarking to Advance Research: A Challenge to Software Engineering. In: *Proceedings of the 25th International Conference on Software Engineering*. ICSE '03. Portland, Oregon: IEEE Computer Society, 2003, pages 74–83. DOI: 10.1109/ICSE.2003.1201189. (Cited on page 18)
- [Traub et al. 2021] J. Traub, P. M. Grulich, A. R. Cuéllar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl. Scotty: General and Efficient Open-Source Window Aggregation for Stream Processing Systems. *ACM Transactional Database Systems* 46.1 (Mar. 2021). DOI: 10.1145/3433675. (Cited on pages 1–3, 5, 10, 17, 23, 24, 74)
- [Traub et al. 2018] J. Traub, P. M. Grulich, A. Rodriguez Cuellar, S. Bress, A. Katsifodimos, T. Rabl, and V. Markl. Scotty: Efficient Window Aggregation for Out-of-Order Stream Processing. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pages 1300–1303. DOI: 10.1109/ICDE.2018.00135. (Cited on page 17)
- [Tucker et al. 2010] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. NEXMark – A Benchmark for Queries over Data Streams DRAFT. In: June 2010. (Cited on pages 31, 33)
- [V. Kistowski et al. 2015] J. v. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao. How to Build a Benchmark. In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE '15. Austin, Texas, USA: Association for Computing Machinery, Jan. 2015, pages 333–336. DOI: 10.1145/2668930.2688819. (Cited on pages 18, 23)
- [Van Dongen 2021] G. van Dongen. *OSPbench Wiki*. Accessed on December 9, 2021. 2021. URL: <https://github.com/Klarrio/open-stream-processing-benchmark>. (Cited on pages 36, 38, 78)
- [Van Dongen and Van den Poel 2020] G. van Dongen and D. Van den Poel. Evaluation of Stream Processing Frameworks. *IEEE Transactions on Parallel and Distributed Systems* 31.8 (Mar. 2020), pages 1845–1858. DOI: 10.1109/TPDS.2020.2978480. (Cited on pages 2, 32, 33, 35, 36)
- [Van Dongen and Van den Poel 2021a] G. van Dongen and D. Van den Poel. A Performance Analysis of Fault Recovery in Stream Processing Frameworks. *IEEE Access* 9 (June 2021), pages 93745–93763. DOI: 10.1109/access.2021.3093208. (Cited on pages 2, 32, 33, 35)
- [Van Dongen and Van den Poel 2021b] G. van Dongen and D. Van den Poel. Influencing Factors in the Scalability of Distributed Stream Processing Jobs. *IEEE Access* 9 (2021), pages 109413–109431. DOI: 10.1109/ACCESS.2021.3102645. (Cited on pages 2, 32, 33, 35, 36, 44, 54, 73)
- [Vonheiden 2021] B. Vonheiden. *Master Thesis Replication Package for: Empirical Scalability Evaluation of Hopping Window Aggregation Methods in Distributed Stream Processing*. Zenodo, Dec. 2021. DOI: 10.5281/zenodo.5764902. (Cited on pages 38, 40, 41, 54)

Bibliography

- [Wang et al. 2021] G. Wang, L. Chen, A. Dikshit, J. Gustafson, B. Chen, M. J. Sax, J. Roesler, S. Blee-Goldman, B. Cadonna, A. Mehta, V. Madan, and J. Rao. Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka. In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD/PODS '21. ACM, June 2021, pages 2602–2613. DOI: 10.1145/3448016.3457556. (Cited on pages 12, 13, 71)
- [Weber et al. 2014] A. Weber, N. Herbst, H. Groenda, and S. Kounev. Towards a Resource Elasticity Benchmark for Cloud Environments. In: *Proceedings of the 2nd International Workshop on Hot Topics in Cloud Service Scalability*. HotTopICS '14. Dublin, Ireland: Association for Computing Machinery, 2014. DOI: 10.1145/2649563.2649571. (Cited on page 19)
- [Zaharia et al. 2012a] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, page 2. (Cited on pages 15, 16)
- [Zaharia et al. 2012b] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. In: *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*. Boston, MA: USENIX Association, June 2012, page 10. (Cited on page 16)
- [Zhang et al. 2021] C. Zhang, R. Akbarinia, and F. Toumani. Efficient Incremental Computation of Aggregations over Sliding Windows. In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. KDD '21. Virtual Event, Singapore: Association for Computing Machinery, 2021, pages 2136–2144. DOI: 10.1145/3447548.3467360. (Cited on pages 12, 15, 74, 78)