CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Low-Latency Anomaly Detection using Stream Processing

Applying the Kolmogorov-Smirnov Test on Streaming Data with Apache Beam

Master's thesis in Computer science and engineering

KATARINA BERGBOM AND MATS HÖGBERG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Low-Latency Anomaly Detection using Stream Processing

Applying the Kolmogorov-Smirnov Test on Streaming Data with Apache Beam

KATARINA BERGBOM AND MATS HÖGBERG

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Low-Latency Anomaly Detection using Stream Processing
Applying the Kolmogorov-Smirnov Test on Streaming Data with Apache Beam
KATARINA BERGBOM AND MATS HÖGBERG

Supervisor: Vincenzo Massimiliano Gulisano, Department of Computer Science and Engineering
Advisor: Victor Risne, Spotify AB
Examiner: Philippas Tsigas, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

Low-Latency Anomaly Detection using Stream Processing
Applying the Kolmogorov-Smirnov Test on Streaming Data with Apache Beam
KATARINA BERGBOM
MATS HÖGBERG
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

To ensure the continuous operation of online services, it is important to be able to quickly detect system failures. This can be done by monitoring metrics, such as the number of logins or errors per hour, for unexpected behaviour. These unexpected behaviours, also known as anomalies, can indicate that something in the system is not working as intended, which makes it important to be able to detect them with low latency. In this thesis, we researched how anomalies can be detected in metrics with low latency using stream processing, a data processing paradigm in which data is processed as continuous streams of events. This thesis was conducted at Spotify, one of the largest audio streaming platforms in the world.

To research low-latency anomaly detection using stream processing, we implemented Harpooner – a stream processing-based counterpart to an existing batch-processing-based anomaly detection system at Spotify. Harpooner analyses metrics in segments, which are subsets of users, and detects anomalies on an hourly basis. Anomalies are detected using the Kolmogorov-Smirnov (K-S) test, a statistical test that can be used to determine if two samples are drawn from the same underlying distribution. Harpooner was implemented using Apache Beam, a programming model for expressing stream processing pipelines. It was implemented in various versions which weighed trade-offs between implementation simplicity, data storage and computational complexity of the K-S test.

Harpooner consists of two parts: a metric calculation part, which is identical in all versions; and an anomaly detection part, which is different in all versions. These parts were evaluated separately using data from Spotify to ensure semantic equivalence between Harpooner and the existing system, and synthetic data to measure their scalability. During evaluation, it was shown that the most efficient anomaly detection part was able to detect anomalies in a metric with 6,000 segments with a latency below 10 seconds when run on a single node on Cloud Dataflow, and that in a real setting the metric calculation part would be the bottleneck of the pipeline. However, if the two parts were deployed as two separate pipelines, our preliminary results indicate that Harpooner would be able to scale to handle the load necessary to do anomaly detection in metrics at Spotify.

Keywords: Stream processing, Anomaly detection, Apache Beam, Streaming systems, Kolmogorov-Smirnov test.

Acknowledgements

We would first and foremost like to thank our company supervisor Victor Risne for his large amount of commitment and help throughout the thesis, along with our manager Lucas Malta for his genuine support. In general, we would like to thank Spotify for providing us with a fun and exciting environment for conducting our thesis. We would also like to thank our Chalmers supervisors Vincenzo Gulisano and Dimitris Palyvos for providing us with their expertise and valuable feedback, which enhanced the quality of the thesis considerably. Finally, we would like to thank our families for their constant support.

Katarina Bergbom and Mats Högberg, Gothenburg, June 2020

Contents

List of Figures

List of Tables

1

Introduction

To ensure the continuous operation of online services, it is important to be able to quickly detect failures so that the causes of the failures can be swiftly investigated and fixed. This can be done by monitoring metrics, such as the number of logins or errors per hour, for unexpected behaviour [39]. However, doing this reliably in an automated fashion is not trivial, as many metrics follow complex patterns, and values that are considered normal one day might be abnormal the next one. Relying on manual data inspection from analysts is not a feasible solution either, as large scale software companies sometimes need to monitor millions of metrics [26].

Metrics such as the number of logins or errors per hour are often calculated by counting events that are generated over time. To quickly detect when the pattern of a metric exhibits unexpected behaviour, i.e. to detect so called *anomalies*, these streams of events need to be processed with low latency. One way to do this is to use a Stream Processing Engine (SPE), such as Cloud Dataflow [14] or Apache Flink [7], that processes events as they arrive in an online fashion. However, this type of data processing adds constraints to the anomaly detection models, making some traditional anomaly detection techniques infeasible in their current form [29].

In this thesis, we researched how stream processing can be applied to detect anomalies in metrics at large software companies. The thesis was conducted at Spotify, one of the largest audio streaming platforms in the world with 286 million monthly active users as of April 2020 [33]. To ensure the continuous operation of their service, Spotify has several anomaly detection systems that monitor different metrics for anomalies. One of these systems is based on batch processing and runs once a day to detect anomalies in certain metrics. For some of these metrics, Spotify believes that there could be a benefit in detecting anomalies with lower latency, making anomaly detection using stream processing in these metrics an interesting research topic.

1.1 Background

As previously mentioned, Spotify has an anomaly detection system based on batch processing that runs once a day to detect anomalies in certain metrics. These metrics are calculated based on events that are being generated when Spotify's service is used, and the values of the metrics are thus naturally influenced by user activity.

Also, since events will continue to be generated as long as Spotify's service exists, these event streams can be considered to be *unbounded*, i.e. of no fixed size.

The system analyses the metrics in *segments*, which are subsets of users. A segment could for example be all users in a specific country or all users with a certain type of device. For each segment, the value of the metric is calculated once per hour. When looking at a specific segment, the metric becomes a time series consisting of one metric value per hour, and it is in these time series that the system is detecting anomalies.

A metric is considered to be anomalous when the values of the metric over a 24-hour time span differs too much from the values of the metric from the preceding 29 days. This difference is determined by using the *Kolmogorov-Smirnov (K-S) test*, which is a statistical test that determines the probability that two samples of data have been drawn from the same underlying distribution. In this case, the K-S test is used to compare the metric values from the two time periods. If the resulting probability is below a certain threshold, the metric values from the 24-hour time span is considered to be anomalous, and an alert is generated by the system.

Spotify believes that there could be a benefit in detecting anomalies with lower latency in some of the metrics monitored by the existing batch-processing-based system. The way we propose doing this is by using *stream processing*, which is a data processing paradigm where data is processed as continuous streams as opposed to in batches. Systems that process data according to this paradigm are designed to process unbounded data streams, which we are dealing with in this case, and switching to stream processing can be a good way to achieve lower latency [5].

1.2 Aim

The aim of this thesis is to investigate how stream processing can be applied at Spotify to detect anomalies in metrics with low latency.

1.3 Problem Definition

As mentioned in Section 1.1, Spotify has an anomaly detection system based on batch processing that runs once a day, and they believe that there could be a benefit in detecting anomalies with lower latency in some of the metrics monitored by this system. In this thesis, we will investigate how a system that does this can be implemented using stream processing.

The metrics monitored by this system are calculated with an hourly granularity, and a metric is considered to be anomalous when the values of the metric over a 24-hour time span differs too much from the values in the metric from the preceding 29 days. The way this is determined is by using the Kolmogorov-Smirnov (K-S) test. In order to keep the same anomaly detection semantics as the existing system, our system will use the same data and the same definition of what constitutes an

anomaly as the existing system. With the hourly granularity of the metrics, the shortest alert interval that can be achieved is hourly. Since we want to achieve the lowest latency possible, hourly detection is what our system will implement. The first research question is thus:

Q1: How can an anomaly detection system with the same semantics as the existing system, but with an hourly alert interval, be implemented using stream processing?

The existing system uses the K-S test to detect anomalies, and this test is difficult to apply efficiently on streaming data [24]. At Spotify, both the number of metrics and the amount of data that is used to calculate the values of the metrics are large and constantly increasing. In order for the implemented system to be able to scale to handle the load at Spotify, we need to find a way to apply the K-S test efficiently when using stream processing. The second research question is thus:

Q2: How can the Kolmogorov-Smirnov test be applied efficiently on a data stream when using stream processing?

To evaluate the efficiency of the system, *throughput* and *latency* will be used. Here, throughput is defined as the number of events that can be ingested per second by the system, and latency is the maximum time it takes for an event to be processed after it has been received. In other words, throughput is a measure of the volume of data that can be processed by the system, and latency is a measure of how long time it takes for the system to process that data.

Our system will use an hourly alert interval, while the existing system uses a daily alert interval. This decrease in the alert interval might create noticeable changes to when the anomalies are found and how many times the system is alerting, which leads us to our third and final question:

Q3: How does hourly detection of anomalies compare to daily detection, in terms of which anomalies are detected and how early the anomalies are detected?

This question concludes the problem definition, as the main aspects of the streaming system has now been investigated according to our aim.

1.4 Limitations

Apache Beam is a programming model that supports the development of both batch and stream processing pipelines. Spotify uses Scio – a Scala API for Apache Beam – for various data processing tasks, and has a well developed infrastructure in place for the development and testing of Scio jobs [27]. The Beam programming model was deemed to be rich enough to express the logic of our system, and was thus chosen for the implementation of our system.

Another limitation is to only look at one of Spotify's currently implemented anomaly detection systems. With possible gains and broad usage at Spotify in mind, the

system using the K-S test was chosen to be the scope for answering question **Q1** and **Q3**.

1.5 Risk Analysis and Ethical Considerations

Anomaly detection in data streams generated by user activity comes with some risks. As streams from a single user's activity reflect what he or she is interested in and how his or her taste varies over time, there might be a personal reason behind an anomaly. Hence, finding anomalies can intrude the user's privacy. In order to prevent privacy intrusions, data from individual users should never be analysed separately, only in group, which is the case throughout this thesis.

This type of abuse of an anomaly detection system could also appear in other areas where monitoring of user behaviour exists. For example, finding anomalies in gaming or drinking behaviour could indicate when an addict tries to stop. Such information could be exploited by selling companies, for example by customizing ads to keep people addicted. However, as there is no new anomaly detection algorithm developed in this thesis, we deem the risk of this content to be misused to be fairly small.

2

Preliminaries

This chapter provides in depth explanations of the concepts used in this thesis. The chapter begins with Section 2.1, which explains what anomaly detection is and describes the Kolmogorov-Smirnov test – the anomaly detection method used in this thesis. Section 2.2 describes what stream processing is and is followed by Section 2.3 which describes Apache Beam – the programming model used in our implementations. Finally, Section 2.4 describes the Google Cloud Platform services which are used for evaluating the implemented system. After reading this chapter, a reader with basic knowledge of computer science should be able to understand the implementation, evaluation and discussion introduced in the subsequent chapters.

2.1 The Kolmogorov-Smirnov Test

An *anomaly* is a point or pattern that does not conform to expected normal behaviour [8], and can indicate negative changes in a system [3]. For this reason, anomaly detection is a useful tool in many industries [6, 8, 19, 36, 37]. However, as it is often difficult to enclose what defines a normal behaviour and adapt to its changes over time, selecting a suitable anomaly detection method can be challenging.

The anomaly detection method used in this thesis is the Kolmogorov-Smirnov (K-S) test [28], which is a traditional data analysis technique. The K-S test exists in two versions: the one-sample version and the two-sample version. The one-sample version evaluates how likely it is that one sample is drawn from a given distribution, and the two-sample version evaluates how likely it is that two samples are both drawn from the same distribution. Some advantages of the K-S test are that the underlying distribution does not need to be known in advance, and there is no need for manual tuning of parameters. Furthermore, it is an unsupervised method, meaning that it can be applied on unlabeled data sets. These advantages mean that the K-S test can be applied to various kinds of data [24], and more importantly, it can be applied on the metrics examined in this thesis.

In this thesis, the two-sample K-S test is used. The two-sample test takes an input of two observed samples and evaluates the null hypothesis, H_0, which states that the two samples were drawn from the same underlying distribution. The output of this test is a p-value, i.e. the probability of achieving a result as least as extreme

as the given result given that H_0 is true. In the case of the two-sample K-S test, a low p-value indicates that the two input samples are most likely not drawn from the same distribution. Below follows a more detailed explanation on how the p-value in the two-sample test is calculated.

Given the two input samples, $X = \{X_1, X_2, \ldots, X_n\}$ and $Y = \{Y_1, Y_2, \ldots, Y_m\}$ with sizes n and m respectively, the first step is to determine their Empirical Cumulative Distribution Functions (ECDFs). Given a value x, an ECDF for a sample returns the percentage of values in the sample that are smaller than or equal to x. The ECDFs for the two samples are denoted as $F_{1,n}(x)$ and $F_{2,m}(x)$, and their formal definitions are stated in Eq. (2.1). Notice that the larger the sample size is, the closer the ECDF should be to the corresponding Cumulative Distribution Function (CDF) of the underlying distribution of the sample.

$$\begin{aligned} F_{1,n}(x) &= \frac{|\{i \mid X_i \leq x\}|}{n} \\ F_{2,m}(x) &= \frac{|\{i \mid Y_i \leq x\}|}{m} \end{aligned} \tag{2.1}$$

Given $F_{1,n}(x)$ and $F_{2,m}(x)$, the *K-S statistic*, $D_{n,m}$, can be computed. $D_{n,m}$ represents the maximal distance at any point between the two ECDFs. It is defined as in Eq. (2.2) and an example of the $D_{n,m}$ given two samples is found in Figure 2.1.

$$D_{n,m} = \sup_x |F_{1,n}(x) - F_{2,m}(x)| \tag{2.2}$$

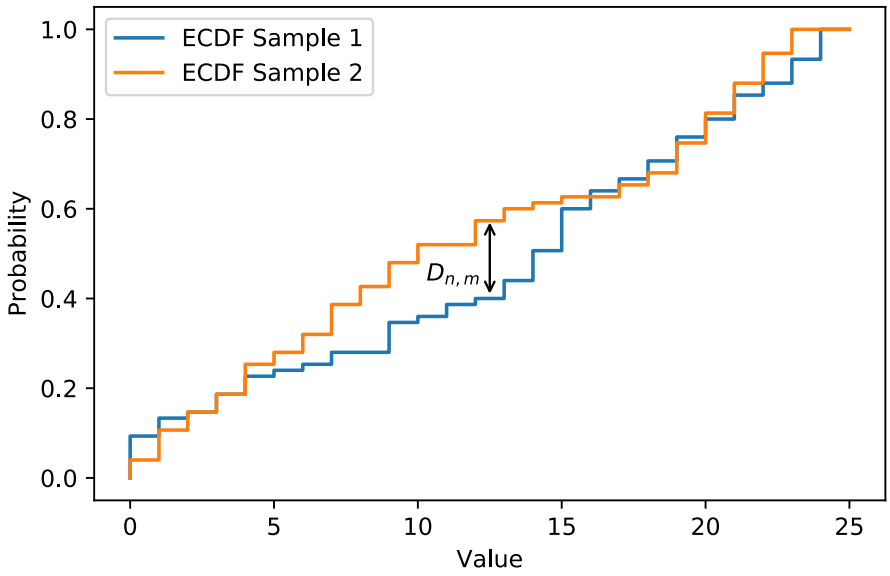


Figure 2.1: The Kolmogorov-Smirnov statistic, $D_{n,m}$, calculated from the Empirical Cumulative Distribution Functions (ECDFs) of two samples.

Lastly, the p-value is calculated. Given d, the specific K-S test statistic for a certain case, and the general K-S statistic $D_{n,m}$, the p-value corresponds to the probability $Pr(D_{n,m} \leq d)$. This probability is defined as in Eq. (2.3). Even though the definition contains an infinite sum, it can be used as a sufficiently accurate approximation for large n and m [32, 31].

$$\lim_{n\to\infty} Pr(D_{n,m} \leq x) = 1 - k\left(d * \sqrt{\frac{mn}{m+n}}\right) \quad \text{where} \quad k(x) = 2\sum_{i=1}^{\infty}(-1)^i e^{-2i^2x^2} \tag{2.3}$$

A low p-value indicates that it is unlikely that the two samples are drawn from the same distribution. When used for anomaly detection in this thesis, one of the samples represents the expected pattern and the other sample represents the values to check for anomalies in. The latter sample is deemed to be anomalous if the p-value from the two-sample test is below a certain threshold.

2.2 Stream Processing

The aim of this thesis is to investigate how stream processing can be applied to detect anomalies in metrics with low latency. To understand what stream processing is, we first need to define what a stream is. In the context of stream processing, a *stream* is an element-by-element view of the evolution of a dataset over time [5]. As an example, the dataset for the metrics dealt with in this thesis consists of the events that are being generated when Spotify's service is used. These events are generated over time, and when viewed one-by-one these events constitute a stream. Datasets, and in extension streams, can either be bounded or unbounded, meaning that they are of either finite or infinite size. The streams of events used to calculate the metrics dealt with in this thesis are examples of unbounded streams.

When using stream processing, datasets are processed as streams, meaning that the elements in a dataset are continuously processed as they arrive. This is in contrast to *batch processing*, which is a more traditional data processing technique, where the dataset is stored and periodically processed in finite chunks, so called *batches*. When processing data in batches, there is naturally a delay in the generation of results, as all the data for a batch needs to arrive before any results can be computed. When using stream processing, this delay can be reduced, as new results can be computed as soon as a new element arrives in the stream.

Systems that are designed to process data using stream processing are called Stream Processing Engines (SPEs), and some examples of SPEs are Storm [35], Flink [7] and Cloud Dataflow [14]. These SPEs are designed to process data streams in parallel on multiple nodes in a distributed system, and thus allow for large volumes of data to be processed with low latency. Note that due to networks being unreliable and latency being non-zero in distributed systems [30], the order in which elements

are processed by an SPE might not be the same as the order in which they were originally generated. This can lead to elements being processed out-of-order, which is an important caveat to have in mind when using an SPE to process data.

As mentioned above, stream processing enables the generation of results with lower latency compared to batch processing. However, doing anomaly detection using stream processing places some requirements on the anomaly detection algorithm. According to Ahmad et al. [3], anomaly detection algorithms for streaming data must be able to, for example:

- make predictions online, i.e. determine if a value in the stream constitutes an anomaly before the next value in the stream is processed;
- learn continuously without having to store the entire stream; and
- quickly adapt to changes in the stream, as the normal behaviour of the stream is likely to change over time.

These requirements mean that some traditional algorithms originally designed for use with batch processing, such as the Kolmogorov-Smirnov test, need to be adapted for use with stream processing [29].

2.3 Apache Beam

Apache Beam, formerly known as *The Dataflow Model*, is a unified programming model that can be used to define both batch and stream processing jobs [4]. In this model, data processing jobs are defined as *pipelines*, and once defined a pipeline can be compiled and executed on different execution engines. In this thesis, we have used Scio – a Scala API for Apache Beam – to implement an anomaly detection system using stream processing.

In the Beam programming model, data processing jobs are defined as pipelines of *transforms*. Transforms are operators that are responsible for reading, modifying and writing data. The intermediate data sets that are created by these transforms are called *collections*, which are immutable and potentially distributed data sets. Collections consist of elements, which can be either simple values or key-value pairs. Depending on the data from which a collection is created, collections can be either bounded or unbounded, i.e. be of a fixed size or be a continuous stream of values. Since collections are immutable they can never change after being created, so transforms that *change* collections rather create new collections where the resulting elements have been created by transforming the elements in the original collection.

An example of a stream processing pipeline written in Scio is found in Figure 2.2. In this example, the number of messages corresponding to errors in a stream are counted, and the resulting counts are output as a new stream. The Cloud Pub/Sub service offered by Google, which is explained in further detail in Section 2.4, is used to both read and write data as streams. To read data from Cloud Pub/Sub, the

```
val sc = ScioContext()

val messages = sc.pubsubTopic[String](inputTopic)
val errors = messages.filter(
    message => message.contains("ERROR"))
val windowedErrors = errors.withFixedWindows(
    Duration.standardMinutes(1))
val errorsPerMinute = windowedErrors.count
errorsPerMinute.saveAsPubsub(outputTopic)

sc.run()
```

Figure 2.2: An example of an Apache Beam pipeline written in Scio. This pipeline counts the number of error messages in a stream.

`pubsubTopic` transform is used, which reads the data as a stream and returns it as an unbounded collection of strings. Then, the `filter` transform is applied to remove all the messages in the stream that does not contain the string `ERROR`. The result of this is a new unbounded collection with the remaining messages. This collection is then windowed into one minute fixed windows using the `withFixedWindows` transform. Windowing is the act of slicing a dataset into smaller, finite chunks, and will be explained in more detailed in Section 2.3.2. The number of error messages in each window are then counted using the `count` transform, resulting in an unbounded collection of counts, which are then output to Cloud Pub/Sub using the `saveAsPubsub` transform. Note that since the messages from Cloud Pub/Sub are read as a stream, this pipeline will continue to process new messages and generate results until it is stopped.

2.3.1 Transforms

In stream processing, transforms can be divided into two categories: stateless and stateful. In stateless transforms, the processing of one event does not affect the processing of other events. This makes it easy to parallelise stateless transforms, as all events in a stream can be processed independently in parallel. In stateful transforms, the result being produced for one event can depend on the previous events in the stream. This means that stateful transforms can not be parallelised in the same way as stateless transforms. In Beam, stateful transforms are applied on a per-key level, meaning that these transforms can be parallelised over keys, but not necessarily more than that.

There are six core transforms in Beam, which can be combined to express all other transforms. Three of these transforms are stateless and three are stateful. The stateless transforms are:

- `parDo`, which applies a user-defined function on each element in a collection. The user-defined function can emit zero, one or more new elements for each

element in the original collection, and a new collection is created with the new elements. The `parDo` transform is the basis for all per-element transforms, such as `map`, which is a `parDo` transform that always emits one output element for each input element, and `filter`, which removes elements from a collection based on some predicate function. As `parDo` transforms process elements individually, they can perform their logic across all elements in a collection in parallel.

- `flatten`, which merges two or more collections with identical types.
- `partition`, which does the opposite of `flatten`. Instead of merging two or more collections, it splits one collection into multiple smaller collections, according to a user-defined partitioning function.

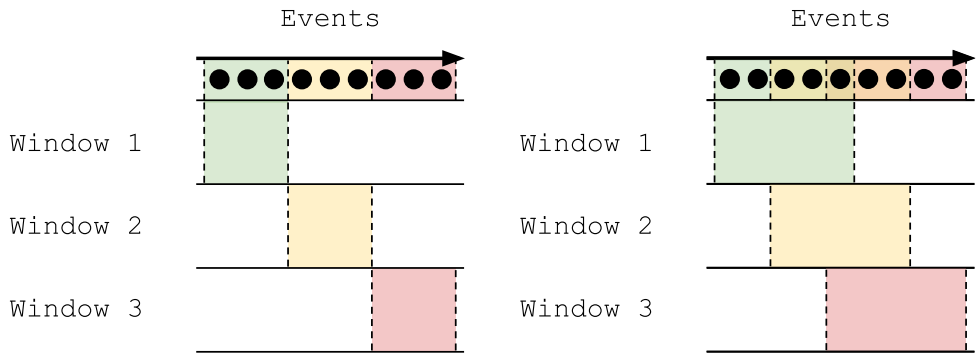
The stateful transforms are:

- `combine`, which combines the elements in a stream using a user-defined function, ultimately returning a single, combined value. This can be used to, for example, calculate the sum of a collection of integers, in which the combiner function would be the addition operator. To allow for parallel computation, the user-defined function needs to be commutative and associative, as this allows partial combinations to be computed in parallel. The `count` transform used in Figure 2.2 is an example of a `combine` transform.
- `groupByKey`, which operates on collections of key-value pairs. This transform collects all values for each key, and emits a new key-value pair for each key where the new value is a list of all the collected values. This can be used as an alternative to `combine`, when the computation can not be easily expressed as a function that is both commutative and associative, such as computing a median.
- `coGroupByKey`, which joins two or more streams using their keys and emits elements with the values in both streams combined. This is similar to the `groupByKey` transform, but instead of grouping together all values for each key in a single stream, it joins two streams using their keys.

Stateful transforms, such as `groupByKey`, `coGroupByKey` and `combine`, can not be applied directly to unbounded collections, as they need access to all elements in the collection in order to produce an output. To get around this, *windowing* can be used.

2.3.2 Windowing

Windowing is the act of slicing a dataset into finite chunks, called *windows*. Beam has support for multiple different windowing strategies. In this thesis, we make use of *fixed windows* (also known as tumbling windows) and *sliding windows*. Visualisations of both of these window types are found in Figure 2.3. Fixed windows have a static window size, and when using fixed windows, each element will be part of exactly



(a) The fixed windowing strategy. **(b)** The sliding windowing strategy.

Figure 2.3: A visualisation of fixed and sliding windows – two windowing strategies supported by Apache Beam.

one window. Sliding windows have a static window size, just like fixed windows, but they also have a sliding period (also known as an advance). This means that, unlike fixed windows, multiple sliding windows can overlap, and each element can be part of multiple windows. A sliding window with size equal to period is effectively a fixed window.

When windowing is used, stateful transforms are applied to and produce results separately for each window. These windows consist of a finite set of values, and consequently when using windowing, transforms that need access to a bounded data set, such as `groupByKey`, `coGroupByKey` and `combine`, can be applied on unbounded collections.

It is common to set the size of windows based on *event-time*, i.e. the time when an event was generated. Since latency is rarely zero when sending data over a network, the event-time of an event will most likely not be equal to the *processing-time* of the event, i.e. the time when the event is processed. For this reason, it can be non-trivial to determine when it is likely that all the data for a window has been received and the results for the window can be emitted.

To deal with this, Beam includes a feature called *watermarks*. A watermark is a metric on how far in event-time the processing of a stream has proceeded. When the watermark passes the end of a window, it is deemed that all of the data up until that point has been received, and the result for the window can be emitted. For some data sources it is possible to create perfect watermarks, meaning that the watermark will never pass the end of a window until all data for that window has been received. However, for some data sources this is not possible, in which case you have to resort to heuristics when calculating the watermarks. Apache Beam uses watermarks by default when windowing based on event-time.

2.3.3 Stateless Transforms Augmented with Custom State

In Beam, stateless transforms can be augmented with custom state [22]. This allows stateless transforms to have a piece of mutable state, such as a data structure, which can be read from and written to when events are processed. One instance of the state is kept for each key-window pair in the collection that is being processed. As of writing, only `parDo` transforms can be augmented with this type of custom state.

One area where this type of augmented `parDo` transforms can be useful is when applying a machine learning model on a stream of data. If the model that is to be applied is stored in the state, it can be continuously trained as new events are processed, meaning that the model can be kept always up-to-date with the newest data. This behaviour could also be achieved by reading the model from a database before each event is processed, and then writing it back when the model has been updated. However, storing the model in the state of the `parDo` transform keeps all the data inside the SPE, which reduces communication overhead.

Even though useful, this type of stateful processing also comes with some drawbacks. Since the state is mutable and can be read from and written to freely during the processing of an event, there is a risk of race conditions occurring if events are processed in parallel. Because of this, Apache Beam forces all events for the same key to be processed sequentially by stateful `parDo` transforms. This means that stateful `parDo` transforms can only be parallelised over keys, and not over all the elements in the stream as normal, stateless `parDo` transforms can. Also, note that even though the events for a key are processed sequentially, Apache Beam does not guarantee that the events are processed in order. This fact needs to be taken into account when implementing a stateful `parDo` transform. Additionally, the state needs to be serialised and deserialised between the processing of each event, which can lead to additional overhead.

2.4 Google Cloud Platform

Google Cloud Platform is a cloud computing platform developed and offered by Google [15]. Spotify runs all of their infrastructure on Google Cloud [20], and thus all the experiments in this thesis were run using this platform. In particular, the Cloud Dataflow and Cloud Pub/Sub services were used.

Cloud Dataflow is a stream and batch processing engine [14], and is one of the engines that Apache Beam pipelines can be executed on. To monitor pipelines run using Cloud Dataflow, the platform exposes a couple of metrics [17]. The values of the metrics for a pipeline can be read from the Cloud Monitoring API [11]. The metrics that we made use of during this thesis were:

- **Throughput**, which is the volume of data being processed at any point in time, measured in either elements/second or bytes/second.
- **System Latency**, which is the current maximum duration that an item of

data has been processing or awaiting processing. This metric is sampled every 60 seconds.

- **Wall Time**, which is an approximate of the total amount of processing time spent on a transform, aggregated across all workers and threads. Due to this aggregation, the value of the Wall Time metric could end up being greater than the total run time of the Cloud Dataflow job.

The metrics provided by Cloud Dataflow provide measurements on either entire pipelines or individual transforms in a pipeline. If more granular data is needed, Cloud Dataflow can also be instructed to save additional profiling data for a pipeline. This data contains wall times of specific methods in the pipeline code, and is sampled during a run. However, the profiling feature in Cloud Dataflow seems to be experimental, and the documentation surrounding it is sparse.

Another service offered on Google Cloud Platform is the messaging service Cloud Pub/Sub [12]. This service delivers *messages*, which consist of data and optional attributes. Through Cloud Pub/Sub, these messages are then delivered as continuous data streams which can be ingested by Cloud Dataflow pipelines. Cloud Pub/Sub itself provides at-least-once delivery, meaning that every message will be delivered at least once. Used together with Cloud Dataflow, a semantic of exactly-once can be achieved as well.

3

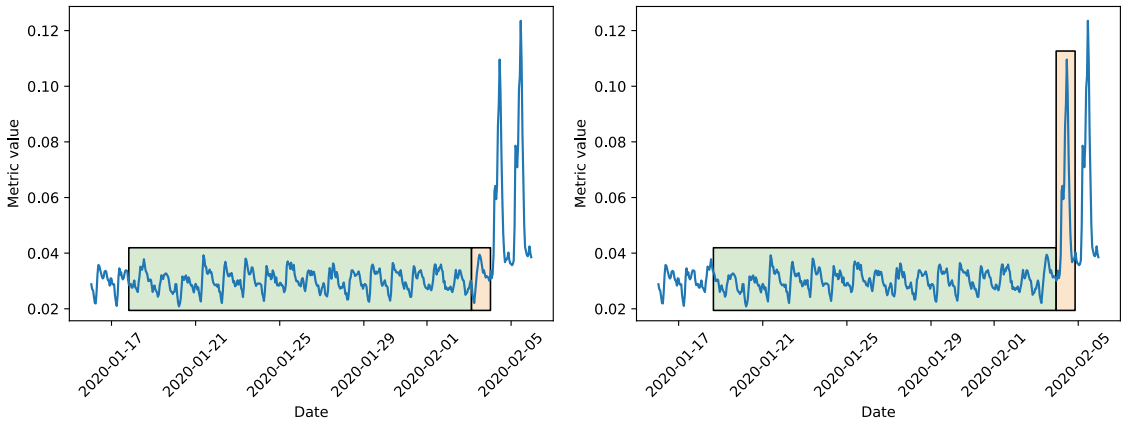
Design and Implementation

In our problem definition, we stated that an anomaly detection system with the same semantics as an existing batch-processing-based system at Spotify, but with hourly instead of daily detection, should be implemented using Scio – a Scala API for Apache Beam. The existing system used the Kolmogorov-Smirnov (K-S) test to detect anomalies in metrics, and a challenge involved in implementing our system was to find a way to apply the K-S test efficiently on streaming data. To weigh the trade-offs between implementation simplicity, data storage and computational complexity of the K-S test, we implemented three versions of our system, which will from here on be referred to as *Harpooner*. The first version of Harpooner used the built-in sliding windows in Apache Beam to produce the samples for the K-S test. In the second version, the sliding windows were replaced with a stateful `parDo` transform that uses custom data structures to reduce the amount of intermediate data in the pipeline. In the third version, two binary search trees were added to the state of the `parDo` transform in order to reduce the computational complexity of the K-S test, at the expense of increased data storage. In this chapter, the designs and implementations of all three versions are explained in detail.

3.1 Requirements of Harpooner

Harpooner should be able to detect anomalies in metrics that are calculated based on events that are being generated when Spotify's service is used. The value of the metric should be calculated with an hourly granularity, where one value per hour should be calculated for each segment in the metric. Anomalies in this metric should be detected using the two-sample K-S test, where the hourly values of the metric from a 24-hour period is used as one of the samples (the 24-hour sample), and the values from the preceding 29 days is used as the other sample (the 29-day sample). When the K-S test is applied using these two samples, the resulting value will be a probability indicating how likely it is that the 24-hour sample and the 29-day sample are drawn from the same distribution. If this probability is below a certain threshold, the 24-hour sample should be deemed to be anomalous. This anomaly detection algorithm is visualised in Figure 3.1, which depicts a metric and the two samples for two consecutive days. In Figure 3.1a, the distributions of values in the two samples are similar, and the 24-hour sample is thus not considered to be anomalous. In Figure 3.1b on the other hand, the distributions of values in the two

samples are not similar, and the 24-hour sample is considered to be anomalous.



(a) For this day, the two samples have similar distributions, meaning that 24-hour sample is not deemed to be anomalous.

(b) For this day, the two samples have distinctly dissimilar distributions. Thus, the 24-hour sample is deemed to be anomalous.

Figure 3.1: A visualisation of how a metric is split into the two samples that are used as input to the Kolmogorov-Smirnov test. This is shown for two consecutive days. The boxes correspond to the 29-day sample and the 24-hour sample. For illustration purposes, the box for the 29-day sample stretches over a time period shorter than 29 days.

To ensure that reliable probabilities are being generated by the K-S test, a limit should be set on how many events that are allowed to be missing from the two samples. The cause for a missing value could for example be that no events were generated during an hour, in which case the metric would have no value for that hour. The limit on the amount of events that are allowed to be missing in our case is 2 for the 24-hour sample and 24 for the 29-day sample, which is the same as the limits used in the existing system.

In Harpooner, the metric should be defined as the ratio between the number of events of two event types. This is a very generic definition, and one that is used by multiple metrics at Spotify. The input to Harpooner should be events of two types, and the output should be alerts for any detected anomalies. It is assumed that the input events have the schema `(key, timestamp)`, where `key` indicates a user segment for the event and `timestamp` indicates when the event occurred. The schema for the alerts that are generated for detected anomalies should also be `(key, timestamp)`, where `key` indicates the segment in which the anomaly occurred and `timestamp` indicates when the anomaly was detected.

3.2 Harpooner 1: Using Sliding Windows to Produce the Samples for the Kolmogorov-Smirnov Test

With the requirements in mind, the pipeline depicted in Figure 3.2 was designed and implemented as the first version of Harpooner. The arrows in this figure represents the flow of data through the pipeline. The inputs to the pipeline are events of two types, here denoted as *e1* and *e2*, and the output are the alerts that are generated for detected anomalies. The pipeline consists of two parts: a metric calculation part and an anomaly detection part.

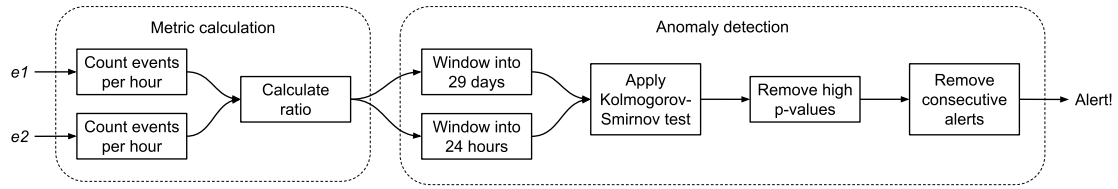


Figure 3.2: An overview of the Harpooner 1 pipeline.

The metric calculation part of the pipeline takes the raw events as input and outputs a segmented metric. This is done by first converting the input streams to key-value streams, where the `key` attribute in the input events is used as the key. These key-value streams are then windowed using fixed event-time windows with a size of one hour, using the timestamp in the `timestamp` attribute. The number of events in each window are then calculated using the `countByKey` transform, which counts the number of elements in a stream on a per key and per window basis. This results in two new streams with hourly counts, which are joined together. Finally, the value of the metric is calculated by taking the ratio of the counts in the two joined streams.

The anomaly detection part of the pipeline detects anomalies in the calculated metric. In order to produce the two samples needed for the K-S test, the stream of hourly metric values is split up into two different streams: one for the 24-hour samples and one for the 29-day samples. These two streams are then windowed using sliding event-time windows, where the size of the window is equal to the size of the sample, i.e. 24 hours for the 24-hour sample and 29 days for the 29-day sample. For both streams, the period of the sliding windows is 1 hour. To extract the samples from the two streams, the `groupByKey` transform is applied to both streams, which groups together all values for the same key in a window and outputs a single output tuple containing all of the values. The streams are then aligned so that the timestamp of a 24-hour sample is equal to the timestamp of the 29-day sample to which it should be compared. Once aligned, the two streams of samples are joined together and used as inputs to a two-sample K-S test. Here, we also apply the limits on the number of events that are allowed to be missing, so that the K-S test is only computed if the samples contain an adequate number of values. For

the calculation of the K-S test, we use the open source implementation found in the Apache Commons Math library [2].

As previously mentioned, the result from the K-S test will be a probability indicating how likely it is that the 24-hour sample and the 29-day sample are drawn from the same distribution. If this probability is below a certain threshold, the 24-hour sample is anomalous and an alert should be generated. To achieve this, a `filter` transform is applied which removes all probabilities that are above a static threshold.

The last operation in the pipeline is the removal of consecutive alerts. The need for this arises from the fact that there is a lot of overlap between the 24-hour samples for two consecutive hours, and if one of the samples is deemed to be anomalous there is a high probability that the next sample will be deemed to be anomalous as well. This would result in multiple alerts for the same anomaly. As all detected anomalies need to be manually investigated, these consecutive alerts are removed so that the only remaining alert is the one for the first hour for which the anomaly was detected. There is a probability that consecutive alerts could correspond to different anomalies, causing Harpooner to not detect some anomalies when consecutive alerts are removed. We discuss how this can be addressed in Section 5.3.

3.3 Harpooner 2: Using Custom Data Structures to Reduce the Amount of Intermediate Data

In Harpooner 1 and the existing system, anomalies are detected by comparing the metric values from a 24 hour time period (the 24-hour sample) to the metric values from the preceding 29 days (the 29-day sample). As each metric value represents one hour of data, only a single value will differ between the samples for two consecutive hours. This is not taken into consideration in Harpooner 1, which uses sliding windows and the `groupByKey` transform to produce the two samples. This combination of transforms will lead to new arrays being created for each sample, which results in a large amount of redundant data flowing through the pipeline.

To reduce the amount of data in the pipeline, we can use the *single-window* strategy. With this strategy, a number of tuples equal to the size of the window are maintained for each key, and the results are computed every time the window is full [9]. In our case, the number of tuples would equal 720 (30 days of hourly metric values), and results would be computed every time a metric value for a new hour has been calculated. Using the single-window strategy can be efficient for supporting stream aggregation when having a sliding window with a large window size and a small period [9]. This is the case in in Harpooner 1, where the sizes of the sliding windows for the 24-hour and 29-day samples are 24 hours and 29 days respectively, and the period for the sliding windows for both samples is 1 hour.

To apply the single-window strategy in Harpooner, the sliding windows for the 24-hour and 29-day samples were replaced with a stateful `parDo` transform which keeps the data for each key in the current window of the stream as internal state and

produces K-S test results when new metric values are available. An overview of the Harpooner 2 pipeline can be seen in Figure 3.3. Apart from the way the K-S test is applied to the stream of metric values, the rest of the pipeline is exactly the same as in Harpooner 1.

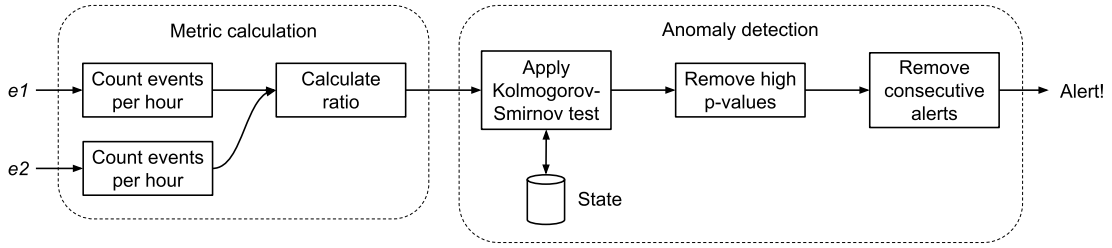


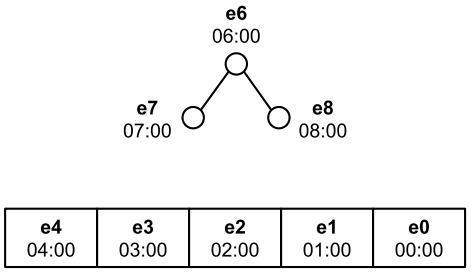
Figure 3.3: An overview of the Harpooner 2 pipeline.

The internal state for the stateful `parDo` transform consists of two data structures: a circular buffer with 720 slots and a min-heap that orders its elements according to their timestamps. The idea is to use the circular buffer as a sliding window, where old events naturally fall out of the buffer when new events are added. Since the metric calculation part of the pipeline always produces a single metric value per key per hour, we know *a priori* that 720 events for a specific key corresponds to exactly 30 days of data. Therefore, a circular buffer of size 720, where events are inserted in order according to their timestamps, has the same semantics as a sliding window with a size of 30 days and a sliding period of 1 hour.

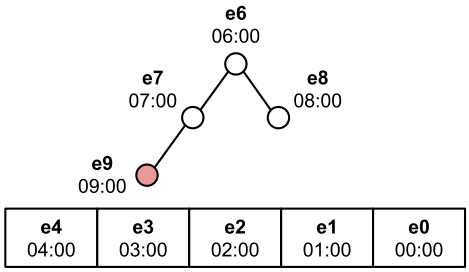
As mentioned in Section 2.3.3, Apache Beam does not guarantee in-order processing of events. Therefore, simply adding events to the circular buffer in the order in which they are processed will not guarantee that the events are inserted in the correct order. To ensure correct ordering, the min-heap is used as a temporary buffer for events. When new events are processed, they are first added to the min-heap, and then *shifted* from the min-heap to the circular buffer in the correct order.

Using these two data structures, the processing of an event can be separated into two operations: *Insert* and *Poll*. The Insert operation adds the event that is being processed to the min-heap, and the Poll operation shifts an event from the min-heap to the circular buffer and computes K-S test results. The Poll operation does this shift in a way that ensures that the events are inserted into the circular buffer in the correct order. When an event is processed, the Insert operation is performed first to insert the event into the min-heap. Then, the Poll operation is repeatedly performed, shifting events from the min-heap to the circular buffer, until no more events can be shifted. This happens either when the min-heap is empty, or when the event with the earliest timestamp in the min-heap is not the event that should immediately follow the last event that was shifted to the circular buffer. In order to be able to quickly determine the latter, the timestamp of the next event that should be shifted to the circular buffer is saved in the state as a separate variable. Whenever a Poll occurs, the timestamp of the earliest event in the min-heap is checked against this timestamp, and only if the two timestamps are equal, the event is shifted.

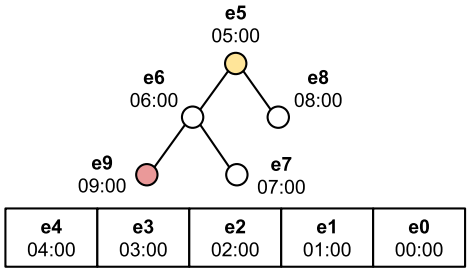
Each time an event is shifted from the min-heap, the circular buffer is split into two arrays: one for the 29-day sample and one for the 24-hour sample. These arrays are then fed into the K-S test to produce a p-value. The p-values that are calculated are the output of the stateful `parDo` transform.



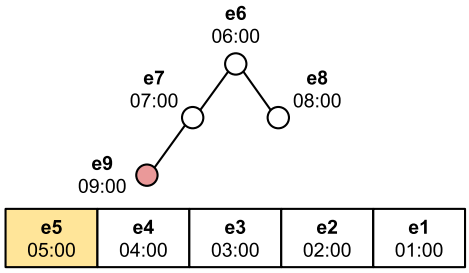
(a) The initial state.



(b) The event with timestamp 09:00 is processed and added to the min-heap.



(c) The event with timestamp 05:00 is processed and added to the min-heap.



(d) The event with timestamp 05:00 is shifted from the min-heap to the window buffer, thus advancing the window by one hour and triggering the computation of the K-S test.

Figure 3.4: An example showing how a circular buffer and a min-heap is used to simulate a sliding window that processes events in order according to their timestamps. In this example, the window size is 5 hours.

To demonstrate how this works, we can look at the example depicted in Figure 3.4. In the initial state, shown in Figure 3.4a, the circular buffer contains events for the time range 00:00 to 04:00, and the min-heap contains events with timestamps 06:00, 07:00 and 08:00. Since the last event in the circular buffer has the timestamp 04:00, the next event to be inserted into the buffer is the event with timestamp 05:00. This event is not in the min-heap, meaning that it has not been processed yet, so the window can not be advanced at this point. In Figure 3.4b, the event with timestamp 09:00 arrives and is added to the min-heap. Since this is not the event that we are waiting for, the circular buffer is left untouched. In Figure 3.4c, the event with timestamp 05:00 arrives. This event is first added to the min-heap and then shifted from the min-heap to the circular buffer, thus advancing the window by one hour and triggering the computation of the K-S test. With the shifting of this event to the circular buffer, there is no longer a gap in timestamps between the

events in the min-heap and the events in the circular buffer, which means that the rest of the events can be shifted as well. This will be done by repeatedly performing the Poll operation until all events in the min-heap have been shifted to the circular buffer.

When processing events in the way described above, special care is needed in order to handle hours without metric values. Simply waiting for the event with the correct timestamp to appear in this case would cause the min-heap to grow indefinitely, and no new p-values would be generated by the pipeline. When using the built-in windows in Apache Beam, this problem is solved through the use of watermarks. If the watermark progresses past the timestamp of a missing metric value, that metric value can be considered to be missing, and the pipeline can be advanced. However, when using a stateful `parDo` transform, we could not find a way to access the watermark of the pipeline. Therefore, we instead set a limit on how much the min-heap can grow. If the size of the min-heap reaches this limit, empty events are inserted into the circular buffer instead of waiting for the event with the correct timestamp to arrive. Empty events are filtered out before the data is fed into the K-S test, and the limits for how many events that are allowed to be missing from the two samples are enforced, meaning that the K-S test is only computed if both samples contain an adequate number of values.

Since events can in theory arrive arbitrarily late, there is no way of knowing if the reason behind a missing event is that it is late or if it has never occurred at all. Therefore, the limit on the size of the min-heap becomes a parameter for deciding the allowed lateness of events. This limit will affect the latency of the pipeline, as for each additional value that is allowed to be added to the min-heap, the latency of the pipeline in the case of missing events increases by up to one hour. However, in a real scenario, it is unlikely that a metric value is more than one hour late. Therefore, it would probably be sufficient to set the limit to 0 or 1, meaning that the allowed lateness of events are 1 or 2 hours, respectively. Additionally, metric values will only be missing if no events were received by the pipeline for an entire hour for a specific segment. In Spotify's case, this is a very unlikely scenario.

In Apache Beam, a serialisation mechanism needs to be provided for the internal state. We used the built in serialisation provided by the `java.io.Serializable` interface, as it provided an easy way to provide serialisation for both the circular buffer and the min-heap.

To analyse the complexity for the processing of an event in Harpooner 2, let n_1 and n_2 be the sizes of the two samples and m be the maximum allowed size of the min-heap. Also, remember that `Insert` is the operation that inserts an event into the min-heap and `Poll` is the operation that shifts events from the min-heap to the circular buffer and computes K-S test results. We will analyse the complexity of these two operations separately.

The complexity of the `Insert` operation is $\mathcal{O}(\log(m))$, which comes from the fact that an event is added to the min-heap. The `Poll` method is a bit more complex. Here, we remove an event from the min-heap ($\mathcal{O}(\log(m))$, add an event to the

circular buffer ($\mathcal{O}(1)$), split the circular buffer into two samples ($\mathcal{O}(\text{Split}(n_1, n_2))$ and apply the K-S test to the two samples ($\mathcal{O}(\text{KS}(n_1, n_2))$). The complexity of the `Split` operation depends on the number of empty events that have been added to the circular buffer. If there are no empty events, the array that is backing the circular buffer can be sent directly to the K-S test, resulting in a complexity of $\mathcal{O}(1)$. If there are empty events, all non-empty events needs to be copied to a new array, which results in a complexity of $\mathcal{O}(n_1 + n_2)$. Harpooner 2 uses an implementation of the K-S test from the Apache Commons Mathematics Library, for which $\text{KS}(n_1, n_2) \in \mathcal{O}(n_1 \log(n_1) + n_2 \log(n_2))$. Thus, the complexity of the `Poll` operation is dominated by the complexity of the K-S test, and the complexity of the entire operation is $\mathcal{O}(n_1 \log(n_1) + n_1 \log(n_2) + \log(m))$.

3.4 Harpooner 3: Using Binary Search Trees to Reduce the Complexity of the Kolmogorov-Smirnov Test

As mentioned in the previous section, the complexity of Harpooner 2 is dominated by the calculation of the K-S test, which has a complexity of $\mathcal{O}(n_1 \log(n_1) + n_2 \log(n_2))$, where n_1 and n_2 are the sizes of the two samples. This complexity comes from the calculation of the K-S statistic. In the implementation used (found in the Apache Commons Mathematics Library), the K-S statistic is calculated by first copying and sorting the arrays for the two samples. Once sorted, the arrays are iterated from the smallest to the highest value, while their Empirical Cumulative Distribution Functions (ECDFs) and the current maximal distance between these ECDFs is continuously computed. For the sorting, the implementation uses the built-in Java method `java.util.Arrays.sort`. This sorting implementation uses a Dual-Pivot QuickSort algorithm, which has a complexity of $\mathcal{O}(n \log(n))$ [1]. The complexity of the calculation of the K-S statistic, and the K-S test as a whole, is dominated by this complexity.

To reduce the complexity of the K-S test, we needed to remove the need for sorting the two samples every time a K-S test result was computed. This can be achieved by maintaining a sorted copy of the metric values as the stream is being processed. One way to do this is to use Binary Search Trees (BSTs). As an in-order traversal of a BST processes the elements of the tree in order, the BSTs can replace the arrays during the calculation of the K-S statistic and consequently remove the need for sorting. In Harpooner 3, we added two BSTs to the internal state of the `parDo` transform: one for the 24-hour sample and one for the 29-day sample. These BSTs keep copies of the values stored in the circular buffer, and are used when calculating the K-S statistic. When implementing these BSTs, we used the TreeSet collection from the `java.util` package. This data structure does not allow duplicates, so to ensure that all the elements in the tree were considered to be unique, we stored both the metric values and their timestamps in the trees. The entire state in Harpooner 3 can be seen in Figure 3.5. Apart from the modified internal state, Harpooner 3 is identical to Harpooner 2.

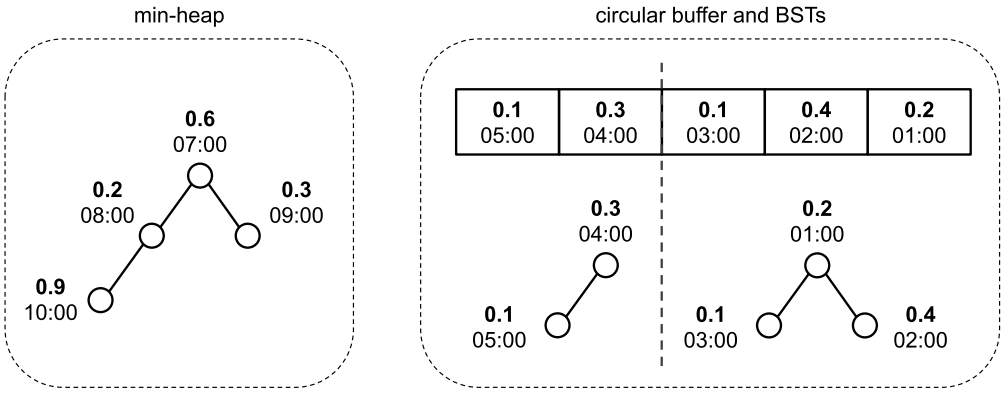
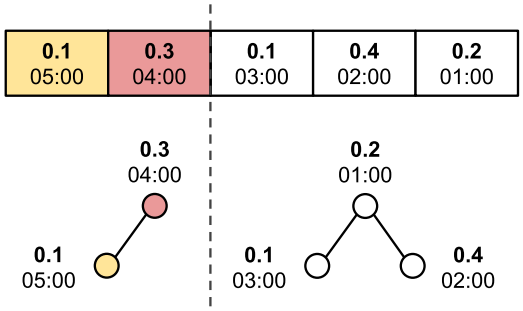


Figure 3.5: An overview of the internal state used in Harpooner 3. The min-heap is shown to the left, and the circular buffer with accompanying Binary Search Trees (BSTs) are shown on the right. The dotted line marks the breakpoint between the two samples in the circular buffer, which in this case have sizes of 2 and 3 hours respectively.

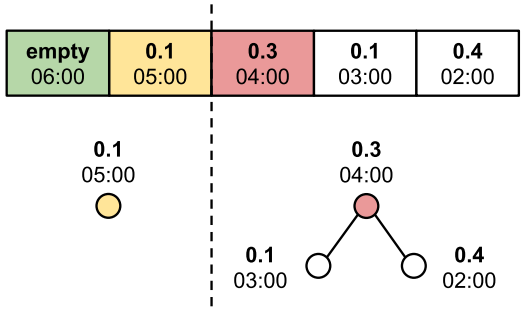
The BSTs are updated each time an event is shifted from the min-heap to the circular buffer. As a result, there are three additional actions when an event is shifted. Firstly, the event that is shifted to the circular buffer is also inserted into the first BST. Secondly, the event that crosses the breakpoint between the two samples due to the insertion is removed from the first BST and inserted into the second BST. Lastly, the event that falls out of the circular buffer due to the insertion is removed from the second BST. This is the case unless any of the three aforementioned events is an empty event. Since empty events should not be included in the samples that are used when calculating the K-S statistic, an empty event should neither be inserted into, nor removed from, the BSTs.

Figure 3.6 illustrates an example of insertions into the circular buffer. In the initial state, shown in Figure 3.6a, the buffer is filled with events with time range 01:00 to 05:00. In Figure 3.6b an empty event with time 06:00 is inserted into the circular buffer. As empty events should not be included in the calculation of the K-S statistic, the event is not added to the first BST. However, the insertion of the empty event into the circular buffer causes the event with timestamp 04:00 to cross the breakpoint between the two samples, so this event is removed from the first BST and inserted into the second BST. The insertion of the empty event into the circular buffer also causes the event with timestamp 01:00 to fall out of the buffer, so it is removed from the second BST. Figure 3.6c shows the next insertion. In this insertion the three events that might cause a change to the BSTs, i.e. the event that is inserted into the buffer, the event that crosses the breakpoint and the event that falls out of the buffer, all contain values. As a result, all three events are inserted into or removed from a BST. The last insertion, seen in Figure 3.6d, causes the empty event to cross the breakpoint. Since empty events are not included in the BSTs, this does not change the state of the BSTs.

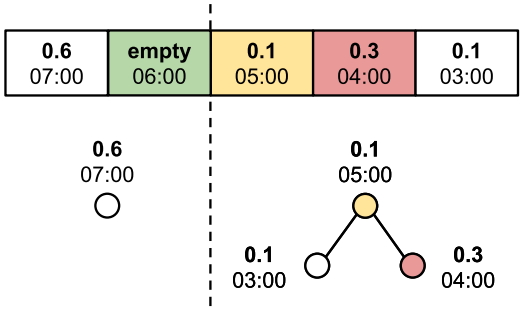
In order to calculate the K-S statistic from the two BSTs, we have adapted the



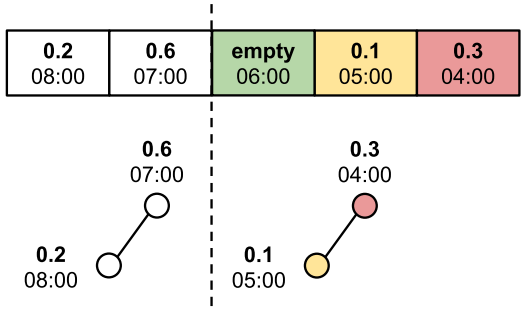
(a) The initial state.



(b) An empty event is inserted in the buffer. It is not inserted in any BST, but causes the event with timestamp 04:00 to be shifted.



(c) The event with timestamp 07:00 is inserted in the buffer and in the first BST. As a result, the event with timestamp 05:00 is shifted.



(d) An event is inserted in the buffer and in the first BST. It causes the empty event to cross the breakpoint, but as the BSTs should not contain empty events, no event is shifted.

Figure 3.6: An example showing how the circular buffer and the two Binary Search Trees (BSTs) are updated when events are shifted from the min-heap to the circular buffer. The dotted line marks the breakpoint between the two samples in the circular buffer. In this example, the window size is 5 hours.

implementation of K-S test from the Apache Commons Mathematics Library. The psuedocode for our algorithm is found in Figure 3.7. In this algorithm, the K-S statistic is calculated by traversing the two BSTs in-order, meaning that the BSTs can be used to calculate the K-S statistic directly without any need for sorting. In each iteration, the smallest of the two values that are currently being traversed is selected, and a distance that corresponds to the current distance of the two ECDFs of the samples is computed. If needed, the maximum distance is updated. In this computation, the distance is not normalized and as a result, the maximum distance is divided by a normalizing factor before it is returned. In the worst case, both trees would need to be traversed in their entirety, leading to a complexity of $\mathcal{O}(n_1 + n_2)$, where n_1 and n_2 denote the sizes of the two samples used for K-S test.

```
def calculate_ks_statistic(x_tree, y_tree):
    n := number of values in x_tree
    m := number of values in y_tree

    x := lowest value in x_tree
    y := lowest value in y_tree

    current_d := 0
    maximum_d := 0

    do while there are values left in both trees:
        z := min(x, y)

        while z == x and values left in x_tree:
            current_d += m
            x = next value in x_tree

        while z == y and values left in y_tree:
            current_d -= n
            y = next value in y_tree

        maximum_d = max(abs(current_d), maximum_d)

    return maximum_d / (n * m)
```

Figure 3.7: Pseudocode for computing the Kolmogorov-Smirnov statistic from two binary search trees.

The addition of the BSTs affects the complexity for the processing of an event. Once again, let n_1 and n_2 be the sizes of the two samples and m be the maximum allowed size of the min-heap. Also, let `Insert` be the operation that inserts an event into the min-heap and `Poll` be the operation that shifts events from the min-heap to the circular buffer and computes K-S test results.

The `Insert` operation has not changed from Harpooner 2, and thus the complexity of this operation remains at $\mathcal{O}(\log(m))$ as motivated in Section 3.3. However, the `Poll` operation has changed. Events are still shifted from the min-heap to the circular buffer with the complexity $\mathcal{O}(\log(m))$, but in addition they are also added or removed from the BSTs. Both adding to and removing from a BST have a complexity of $\mathcal{O}(\log(n_1))$ for sample one and $\mathcal{O}(\log(n_2))$ for sample two. Since the calculation of the p-value given the K-S statistic is constant, the complexity of the K-S test is dominated by the calculation of the K-S statistic. As shown above, the complexity of calculating the K-S statistic from the BSTs is $\mathcal{O}(n_1+n_2)$. Summarized, the total complexity of the `Poll` operation in Harpooner 3 is $\mathcal{O}(n_1 + n_2 + \log(m))$, which is lower than in Harpooner 2. However, the reduction in complexity comes with a performance trade-off in terms of memory, as each value now needs to be

stored twice in the internal state.

4

Evaluation

In the Implementation chapter, we described the implementation of Harpooner – a stream-processing-based anomaly detection system. This system used the same anomaly detection semantics as an existing batch-processing-based anomaly detection at Spotify, but detected anomalies on an hourly instead of a daily basis. In this chapter, we show how the semantics and the scalability of Harpooner were evaluated. In addition, to determine how hourly detection of anomalies compares to daily detection, we compare the alerts generated by Harpooner to the alerts generated by the existing system for a metric at Spotify.

4.1 General Setup

As described in Section 3.2, Harpooner has two parts: a metric calculation part and an anomaly detection part. To make it easier to evaluate the semantics and the scalability of the different versions of Harpooner, the two parts were evaluated separately. We argue that evaluating the two parts separately will yield trustworthy results for both semantics and scalability since: (1) the input to the anomaly detection part is the output of the metric calculation part, meaning that if the semantics of both parts are shown to be correct, the semantics of the entire pipeline should be correct as well; and (2) the performance of the entire pipeline can be derived from the performance of the two individual parts.

The metric calculation part is identical in all three versions of Harpooner, whereas the anomaly detection part is different in all three versions. Therefore, four pipelines needed to be evaluated in total: one metric calculation pipeline and three anomaly detection pipelines.

All evaluation experiments described in this chapter were run on Cloud Dataflow, using a single worker of type n1-standard-4 (the default worker type for streaming Cloud Dataflow jobs) with disk type pd-ssd. This machine type has 4 vCPUs and 15 GB of memory [13]. Cloud Pub/Sub was used for delivering data to the pipelines. The versions of the software used were Scio 0.8.2, Scala 2.12.10 and Apache Beam SDK for Java 2.19.0.

4.2 Semantics

In this section, we begin with explaining how the first criteria of Harpooner was evaluated: having the same anomaly detection semantics as the existing system. Thereafter, we present the results from this evaluation. Lastly, to examine the effects of hourly compared to daily detection of anomalies, we compare the alerts generated by Harpooner to the alerts generated by the existing system.

4.2.1 Data Set

The anomaly detection semantics of Harpooner were evaluated using a metric from Spotify. Due to confidentiality concerns, we cannot disclose any details regarding this metric, but for our purposes it is enough to say that it is defined as the ratio between the hourly counts of two event types. The events of both types contain, along with other attributes, the timestamp of when the event occurred and a key representing a segment for the event. There are 4 segments in total, which will from here on be denoted as *Segment 1*, *Segment 2*, *Segment 3* and *Segment 4*. The value of the metric for Segment 1 during a time period of 90 days, between 2020-01-01 and 2020-03-30, is visualized in Figure 4.1.

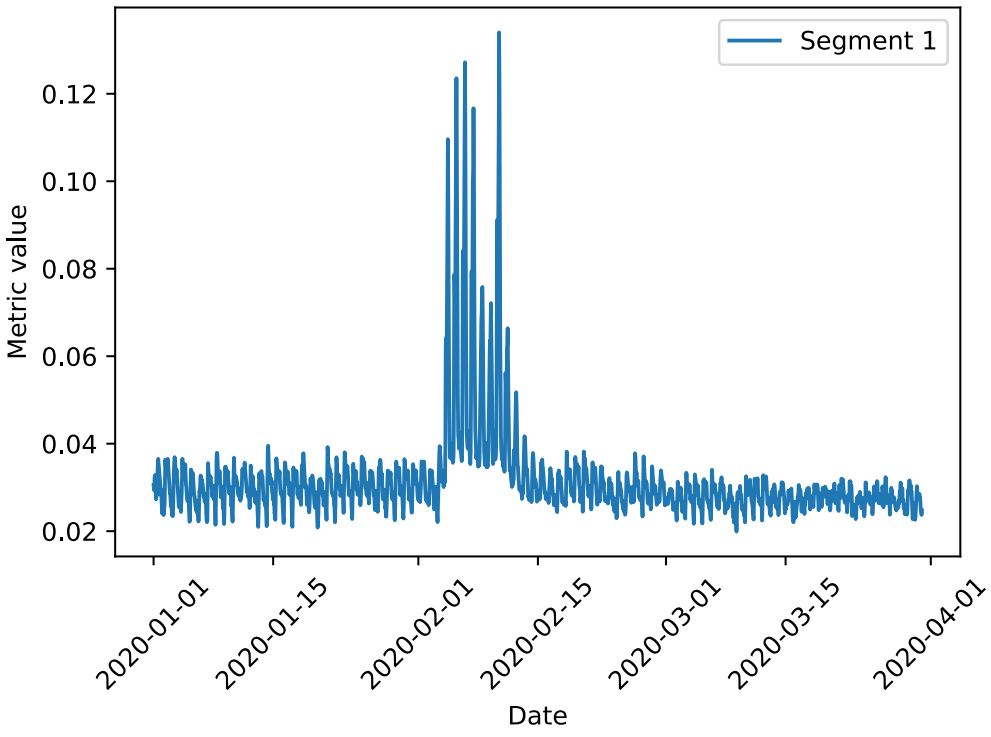


Figure 4.1: The value of the metric used when evaluating the semantics of Harpooner. The value of the metric is shown for a single segment, denoted as *Segment 1*, for the time period 2020-01-01 – 2020-03-30.

The metric followed a regular pattern during most of this time period, but it experienced a series of spikes between 2020-02-04 and 2020-02-12. Figure 4.2 shows the

value of the metric for Segment 1 during a 28 day period prior to the spikes, and gives a better sense of how the metric behaves when following its normal pattern. In this graph, we can see that the metric has a clear daily and also some weekly seasonality. Although not visible in the graphs, there is also a possibility that the metric has some monthly seasonality.

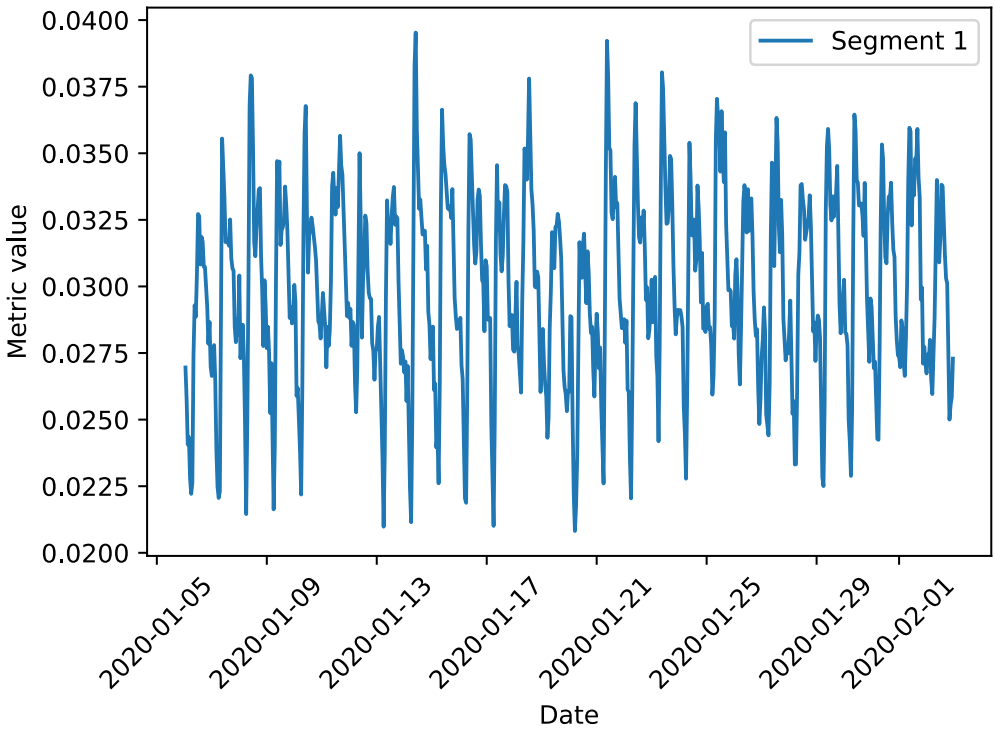


Figure 4.2: The value of the metric used when evaluating the semantics of Harpooner. The value of the metric is shown for a single segment, denoted as *Segment 1*, for the time period 2020-01-06 – 2020-02-02.

The data that was made available to us for this metric was the raw events for both event types, along with precomputed per-segment hourly event counts. We also had access to the daily p-values and the alerts generated by the existing anomaly detection system for this metric.

4.2.2 Metric Calculation

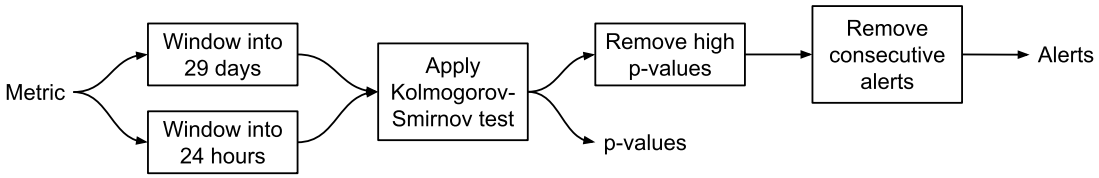
When evaluating the semantics of the metric calculation part of Harpooner, the input data needed to include a sufficient amount of diverse values. With the clear daily seasonality of the metric, we decided that 3 days of data with all four segments would be enough, as this corresponds to 288 metric values within a broad numeric range. As a result, the raw events for the time period 2020-01-01 – 2020-01-03, i.e. 3 days of data, was used during evaluation. To be able to verify the correctness of the results, the metric values were also computed separately using the precomputed hourly counts made available to us.

The result of the experiment was 288 hourly metric values, one for each segment

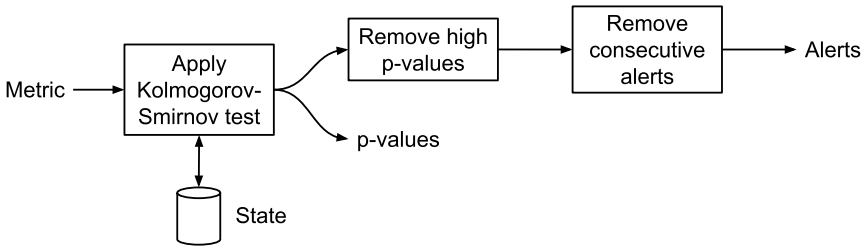
for each hour of data. These values were identical to the metric values that were computed from the precomputed hourly counts.

4.2.3 Anomaly Detection

The metric that was used when evaluating the anomaly detection semantics of Harpooner followed daily, weekly and possibly monthly patterns, all of which could influence the output of the Kolmogorov-Smirnov (K-S) test. When evaluating the semantics of the anomaly detection parts of Harpooner, we deemed it necessary to evaluate a monthly pattern multiple times, and thus we needed data which gave us results from at least 2 months. Due to the way the K-S test is applied in Harpooner, 29 days of data needs to be ingested before any results are produced. Consequently, a total of 3 months of evaluation data was required to give results from 2 months. Hence, for the evaluation we used the metric values for the time period 2020-01-01 – 2020-03-30, i.e. 90 days of data, which were computed using the precomputed hourly counts made available to us.



(a) Flowchart for the pipeline used when evaluating the semantics of Harpooner 1.



(b) Flowchart for the pipelines used when evaluating the semantics of Harpooner 2 and 3. As the only difference between these two versions is the way state is used in the transform that applies the Kolmogorov-Smirnov test, the flowcharts are identical for the two versions.

Figure 4.3: Flowcharts of the modified anomaly detection pipelines that were used when evaluating the semantic equivalence between Harpooner and the existing system. The pipelines were configured to output all computed p-values, in addition to the generated alerts.

To evaluate the semantic equivalence of the anomaly detection parts of Harpooner and the existing system, we wanted to compare the output from the Harpooner pipelines to the output of the existing system, given the same input data. The output from Harpooner is the alerts that are generated when an anomaly is detected, and

since anomalies in metrics are rare, only comparing alerts would not be sufficient. Therefore, we configured the Harpooner pipelines to output all computed p-values, in addition to the generated alerts. Overviews of the modified pipelines for the anomaly detection parts of Harpooner can be seen in Figure 4.3. Harpooner computes p-values every hour, but the existing system only does so once every day at midnight. Therefore, only the p-values for the midnight hour every day could be compared between the systems.

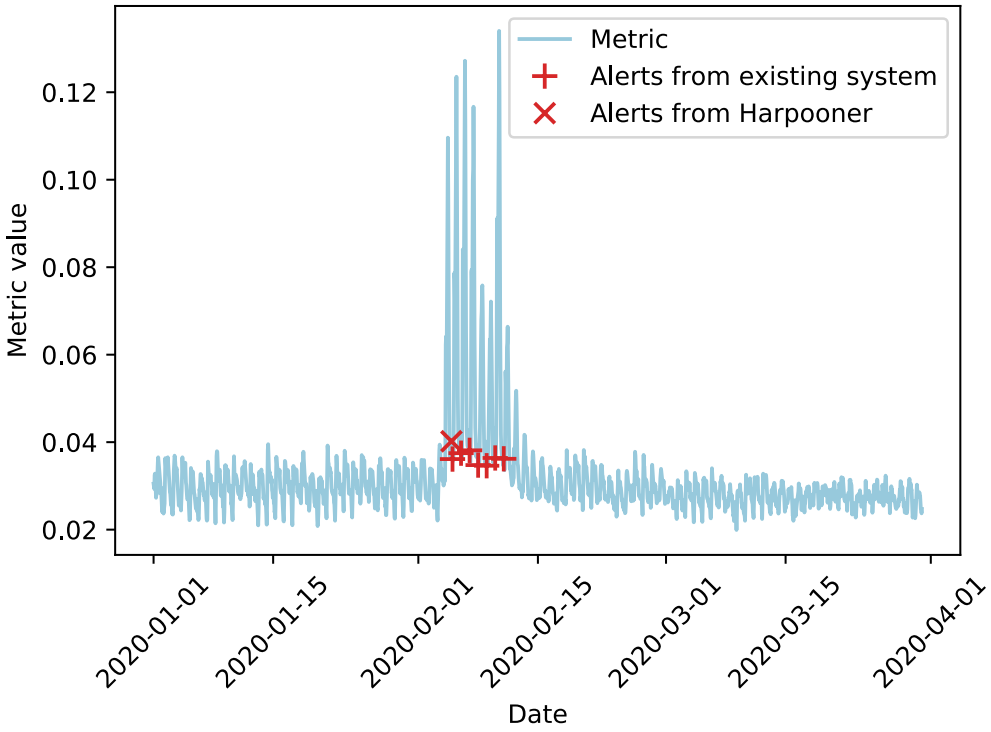
The p-values generated by the modified pipelines were compared to the p-values generated by the existing system for our metric. When compared, these p-values were shown to be equivalent for all four systems – the three versions of Harpooner and the existing system. In addition, the p-values generated for all other hours of the day were identical in all three versions of Harpooner. We also checked the correctness of a selection of these p-values by manually comparing them to p-values computed separately using the K-S test. Here, all the selected p-values were shown to be correct.

In order to answer research question **Q3** – *How does hourly detection of anomalies compare to daily detection, in terms of which anomalies are detected and how early the anomalies are detected?* – we compared the alerts generated by Harpooner, which does hourly detection, to the alerts generated by the existing system, which does daily detection. The alerts generated by both systems for our metric can be seen in Table 4.1.

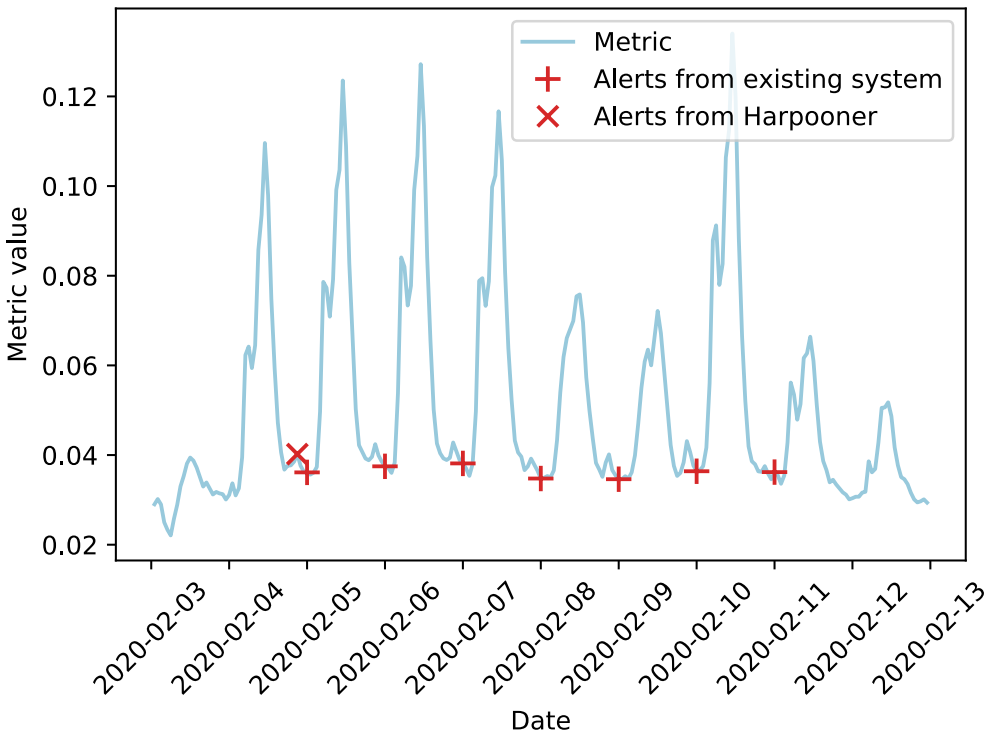
Segment	**Existing system**	**Harpooner**	**Time difference**
Segment 1	2020-02-05 00:00 2020-02-06 00:00 2020-02-07 00:00 2020-02-08 00:00 2020-02-09 00:00 2020-02-10 00:00 2020-02-11 00:00	2020-02-04 21:00	−3 hours
Segment 2	2020-02-27 00:00	2020-02-26 12:00	−12 hours
Segment 3	2020-02-07 00:00	2020-02-07 00:00	±0 hours

Table 4.1: The alerts generated by Harpooner and the existing system.

For Segment 1, the existing system generated 7 alerts in total on seven consecutive days between 2020-02-05 and 2020-02-11, while Harpooner generated a single alert at 2020-02-04 21:00, 3 hours earlier than the first alert from the existing system. The metric for Segment 1, together with the alerts generated by both systems for this segment, is visualized in Figure 4.4a, which shows the entire time period, and Figure 4.4b, which shows a zoomed in view on the alerts that were generated. Notice how both the alert generated by Harpooner and the seven alerts generated by the existing system were caused by a change in the metric that occurred on 2020-02-04. If consecutive alerts would not have been removed in Harpooner, 152 alerts would have been generated in total for Segment 1, one for each hour between 2020-02-04 21:00 and 2020-02-11 04:00.



(a) Data between 2020-01-01 and 2020-03-30, which was the full time span that was used during evaluation.



(b) Data between 2020-02-03 and 2020-02-13.

Figure 4.4: Graphs showing the metric that Harpooner was evaluated on, together with the alerts that were generated by both Harpooner and the existing system. The data for one of the segments, here denoted as *Segment 1*, is shown.

For Segment 2, both systems generated a single alert. The existing system alerted at 2020-02-27 00:00 and Harpooner alerted at 2020-02-26 12:00, 12 hours earlier than the existing system. If consecutive alerts would not have been removed, Harpooner would have generated 13 alerts for Segment 2, one for every hour between 2020-02-26 12:00 and 2020-02-27 00:00.

For Segment 3, both systems generated a single alert at 2020-02-07 00:00. If consecutive alerts would not have been removed, Harpooner would have generated 18 alerts for Segment 3, one for every hour between 2020-02-07 00:00 and 2020-02-07 17:00. None of the systems generated any alerts for Segment 4.

All in all, Harpooner detected the same anomalies as the existing system, and did so earlier or at the same time.

4.3 Scalability

In this section, it is explained how the scalability of the three different versions of Harpooner was evaluated, and the results of this evaluation is presented. The scalability of the pipelines was evaluated in terms of *throughput* and *latency*, where throughput is defined as the number of events that can be ingested per second by the system, and latency is the maximum time it takes for an event to be processed after it has been received. Latency is thus a measurement of how long it takes for a potential anomaly to be detected from when a new metric value is available. To ensure that anomalies were detected with low latency, we required the average latency of Harpooner to be below 10 seconds. The scalability of the pipelines was also evaluated in terms of how many segments the systems could handle. In Harpooner, the `key` attribute is used to represent a segment, and therefore we will from here on use the term *keys* instead of *segments* when speaking of the scalability of the system.

To make it easy to vary the number of keys in the data, we opted for using synthetic data when evaluating the scalability of Harpooner. This synthetic data was designed to mimic the data that would have been processed by the pipelines in a real setting. The specifics of this design for the metric calculation and anomaly detection parts of Harpooner are described in detail in the sections for the respective parts. One disadvantage of using synthetic data is that this data might not be representative of real data. However, since the semantics of Harpooner were evaluated using real data, we consider the usage of synthetic data to be sufficient when evaluating the scalability.

To reduce the time it took to conduct experiments, the synthetic data was generated so that data for 1 hour of event time was fed to the pipelines every second, meaning that 30 days of data could be generated and processed by the pipelines in 12 minutes. Speeding up the data meant that the load on the pipelines was larger during evaluation than it would have been in a real setting. This means that the results from the experiments are not the exact performance limits of the system, but rather represent lower bounds on performance.

To ensure that more than 30 days of data had been ingested by the pipeline, thus fully saturating all windows, and that the pipeline had stabilized before any measurements were taken, all experiments were run with a warm up period of 20 minutes. To get reliable results, measurements were taken during a 10 minute period after the warm up period had ended. This allowed for 25 days of data to be processed while measurements were taken.

To determine the throughput and latency of the pipelines, the Throughput and Latency metrics from Cloud Dataflow were used. As described in Section 2.4, the Throughput metric is defined as the volume of data being processed by the pipeline at any point in time, and the Latency metric is defined as the current maximum duration that an item of data has been processing or awaiting processing. One thing to note is that throughput in Cloud Dataflow is measured on a per-transform basis. In our experiments, we considered the throughput of a pipeline to be the number of events being processed per second by the transform that ingests data from Cloud Pub/Sub. With this consideration, the Throughput and Latency metrics from Cloud Dataflow conformed to our definitions of throughput and latency. The values of the metrics were read from the Google Cloud Monitoring API, and for each configuration, the average throughput and latency for the five experiments was used.

4.3.1 Metric Calculation Setup

As described in Section 4.1, the metric calculation part of Harpooner was separated from the anomaly detection part and the two parts were evaluated separately. When evaluating the performance of the metric calculation part, the synthetic data was designed to mimic the raw events being generated by real usage of the Spotify platform. We used two parameters when generating this data: m, which represents the number of events being generated for each hour of event time; and k, which represents the number of keys in the data.

m (events/hour)	k_1	k_2	k_3	k_4	k_5
1	1	1,375	2,750	4,125	5,500
10	1	375	750	1,125	1,500
100	1	38	75	113	150
1,000	1	5	10	15	20

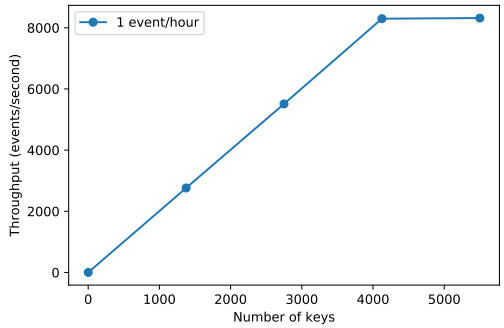
Table 4.2: The values of m (the number of events per hour of event time) and k (the number of keys) that were used when evaluating the scalability of the metric calculation part of Harpooner.

Each message in the data consisted of two attributes: `key` and `timestamp`, where `key` was an integer between 0 and k, and `timestamp` was the simulated event time. To prevent the value of the metric from being exactly 1 all the time, a number of messages in the range $[0.75m, 1.25m)$ was generated per key per hour of event time. The value of the multiplier of m was chosen at random for each key and hour. Since the synthetic data was generated so that data for 1 hour of event time was generated

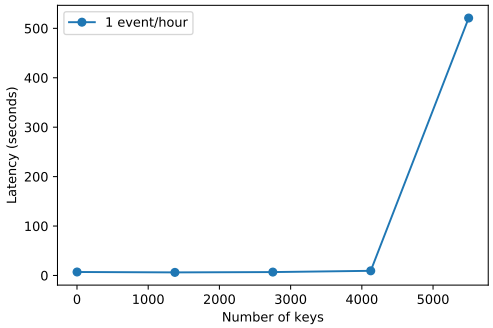
per second, and the metric calculation part of the pipeline takes two event streams as input, the total rate at which messages were generated was approximately $2km$ messages per second.

The pipeline was tested with 4 different values of m, with 5 different values of k for each value of m, resulting in a total of 20 configurations. These values are shown in Table 4.2. The values were chosen based on the maximum throughput that the pipeline were able to handle.

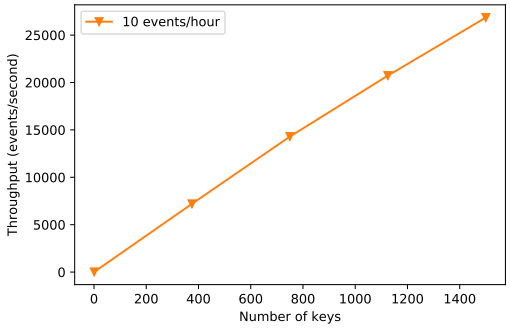
4.3.2 Metric Calculation Results



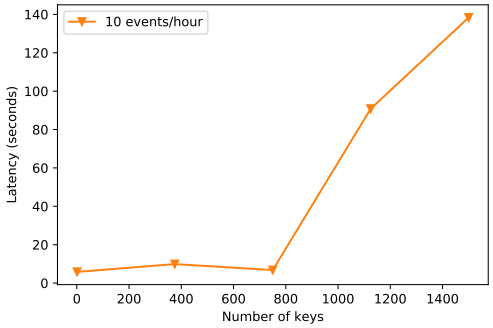
(a) The average throughput with 1 event/hour.



(b) The average latency with 1 event/hour.



(c) The average throughput with 10 events/hour.

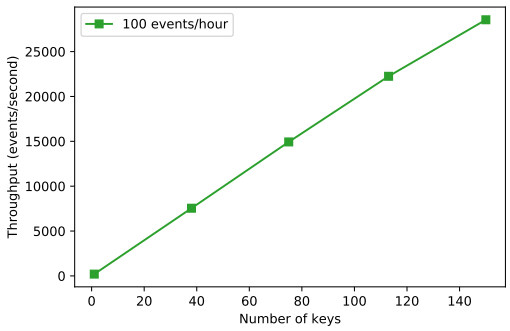


(d) The average latency with 10 events/hour.

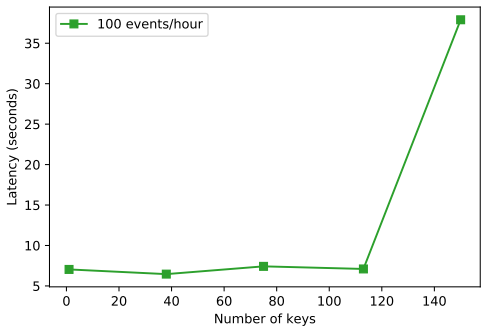
Figure 4.5: The average throughputs and latencies of the metric calculation part of Harpooner during the scalability evaluation experiments. This figure includes the graphs for when events were being generated at a rate of 1 respectively 10 event(s)/hour.

The average throughput and latency of the metric calculation part of Harpooner during the experiments can be seen in Figure 4.5 and Figure 4.6. The throughput increased linearly as the number of keys increased, until a certain point where the pipeline was not able to keep up with the rate at which events were being generated. With 1 and 1,000 event(s)/hour, this point can be seen clearly in the graphs. With

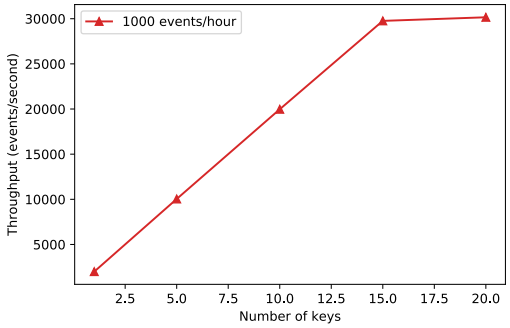
10 and 100 events/hour, the deviation from the linear increase in throughput is also present, although more subtle. If the pipeline would have been tested with a higher number of keys this deviation would most likely have been more apparent. However, this was deemed to be unnecessary as even a small difference between the rate at which events are generated and the throughput of a pipeline will lead to a growing backlog of unhandled events. As a result, it will take longer and longer for new events to be processed, leading to rising latencies.



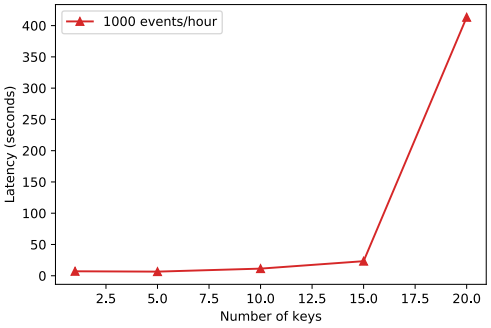
(a) The average throughput with 100 events/hour.



(b) The average latency with 100 events/hour.



(c) The average throughput with 1,000 events/hour.



(d) The average latency with 1,000 events/hour.

Figure 4.6: The average throughputs and latencies of the metric calculation part of Harpooner during the scalability evaluation experiments. This figure includes the graphs for when events were being generated at a rate of 100 respectively 1,000 event(s)/hour.

The average latency stayed fairly constant at around 6-8 seconds until the number of keys reached the point where the pipeline was not able to keep up with the rate at which events were being generated. This was expected, because, as mentioned above, a throughput that is lower than the rate at which events are being generated will lead to an increase in latency.

The maximum number of keys that were able to be handled with a latency below 10 seconds for the different number of events/hour can be seen in Table 4.3.

With 1 event/hour, 2,750 keys could be handled; with 10 events/hour, 750 keys could be handled; with 100 events/hour, 113 keys could be handled; and with 1,000 events/hour, 5 keys could be handled. Due to the coarse granularity of the k values that the pipeline was evaluated with, these values represent approximative rather than true performance limits of the pipeline.

m (events/hour)	k (number of keys)	Average throughput (events/second)	Average latency (seconds)
1	2,750	5,511	6.86
10	750	14,295	6.78
100	113	22,250	7.56
1,000	5	10,037	6.60

Table 4.3: The maximum number of keys that the metric calculation part of Harpooner was able to handle with a latency below 10 seconds.

As the metric calculation part of Harpooner is fairly simple compared to the anomaly detection part, we will not analyse these results further. Instead, we will continue by describing the scalability evaluation of the anomaly detection parts of the three versions of Harpooner.

4.3.3 Anomaly Detection Setup

When evaluating the scalability of the anomaly detection part of Harpooner, the synthetic data was designed to mimic the output from the metric calculation part of the pipeline. When generating this data, we used a parameter k to represent the number of keys in the data.

Each message in the data consisted of three attributes: `key`, `timestamp` and `value`, where `key` was an integer between 0 and k, `timestamp` was the simulated event time and `value` was a floating point value between 0 and 1 chosen at random from a uniform distribution. One message per key per hour of event time was generated, which corresponds to the granularity at which the metric calculation part of the pipeline generates values. Since the synthetic data was generated so that 1 hour of event time was generated per second, the total rate at which messages were published for all keys was k messages per second.

The K-S test used in Harpooner detects anomalies by determining the possibility that two samples are drawn from the same underlying distribution. In the synthetic data, we chose all values from the same distribution, meaning that the data would most likely not contain any anomalies. The effect of this would be that no p-values make their way past the transform that removes high p-values from the stream, meaning that the subsequent transforms do not do any work during the experiments. However, we do not consider this to be a problem, as the majority of the work in the pipeline happens upstream from this transform, with the windowing of the stream of metric values and the computation of the K-S test. Also, in a real setting, anomalies

are very rare. During each experiment the pipelines processed 25 days of data, and in a real setting it is not unlikely that no anomalies would be found during a time period of the same length.

Each pipeline was tested with 5 different values of k. These values are shown in Table 4.4 and were chosen based on the maximum throughput that the pipelines were able to handle.

Pipeline	k_1	k_2	k_3	k_4	k_5
Harpooner 1	1	50	100	150	200
Harpooner 2	1	3,000	6,000	9,000	12,000
Harpooner 3	1	2,000	4,000	6,000	8,000

Table 4.4: The numbers of keys $(k_1, \ldots, k_5)$ that were used when testing the performance of the anomaly detection part of the different versions of Harpooner.

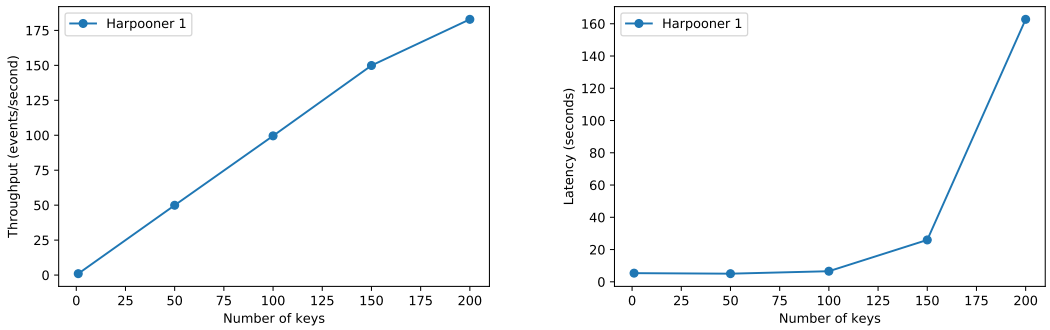
To find bottlenecks in the pipelines we used the Wall Time metric in Cloud Dataflow, which is a measurement that can be used to determine how much time is spent on different transforms in a pipeline. For Harpooner 1, we used the average value of the Wall Time metric from the runs where $k = 150$. Unfortunately, it was only possible to get the Wall Time metric for the entire duration of the experiments, which included the 20 minute warm up period. While it would have been better to only use the Wall Time metric for the last 10 minutes of the experiments, we still believe that the value of the metric for the entire duration of the experiment can provide meaningful insights into the performance of the pipeline.

For Harpooner 2 and 3, both the windowing and the application of the K-S test have been replaced by a single stateful `parDo` transform. The Wall Time metric in Cloud Dataflow only shows the wall time for entire transforms, which means that this metric could not be used to find bottlenecks inside the stateful `parDo` transform. Therefore, for Harpooner 2 and 3 we ran additional profiling experiments with $k = 4{,}000$. The reason for running these experiments separately was to ensure that the saving of profiling data did not have a negative impact on the performance of the pipelines. The profiling data was analysed with pprof [16]. As mentioned in Section 2.4, the profiling feature of Cloud Dataflow seems to be experimental, and for that reason we do not fully trust the results that it generates. However, as with the Wall Time metric for Harpooner 1, we still believe that the profiling results could provide meaningful insights into the performance of the pipelines. Also note that Cloud Dataflow only saves sampled profiling data, which means that the wall times in the profiling data can not be directly compared to the wall times fetched from the Cloud Monitoring API.

Apart from the saving of profiling data, the setup for the profiling experiments were identical to the setup for the other experiments.

4.3.4 Anomaly Detection Results

The average throughput of Harpooner 1 during the experiments is found in Figure 4.7a. Here, it can be seen that the throughput scaled linearly as the number of keys increased, which was expected as the rate at which messages were being published was equal to the number of keys. However, with 200 keys the average throughput was 183 events/second instead of the expected 200 events/second. This shows that with 200 keys, Harpooner 1 was not able to keep up with the rate at which new events were being generated.



(a) A graph showing the average throughput of Harpooner 1 during the experiments.

(b) A graph showing the average latency of Harpooner 1 during the experiments.

Figure 4.7: The average throughput and latency for Harpooner 1 for five different numbers of keys.

The average latency of Harpooner 1 during the experiments can be seen in Figure 4.7b. Here, we can see that the latency stayed fairly constant at around 6-8 seconds up to 100 keys. For 150 keys the latency increased to 26 seconds, and for 200 keys the latency increased to 163 seconds. As we saw in Figure 4.7a, Harpooner 1 was not able to keep up with the rate at which new events were being generated when the number of keys was set to 200, which in turn also caused the latency to increase as it took longer and longer for new events to be processed by the pipeline. The maximum number of keys that Harpooner 1 was able to handle with a latency below 10 seconds was 100.

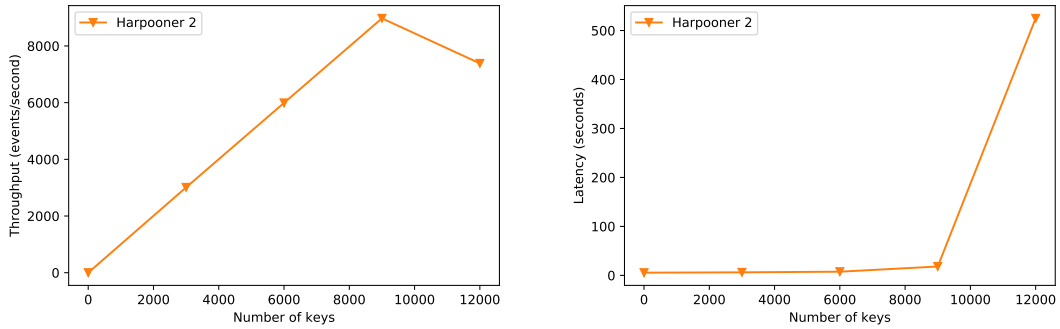
Transform	**Wall time (hh:mm:ss)**
`groupByKey` (29-day sample)	28:06:27
`join`	0:45:42
`groupByKey` (24-hour sample)	0:31:03
Computation of K-S test	0:18:38

Table 4.5: The transforms in the Harpooner 1 pipeline, ordered by decreasing wall time. The table only shows transforms with wall times longer than 5 minutes.

The average values of the Wall Time metric for Harpooner 1 when $k = 150$ can be seen in Table 4.5. This table only includes the transforms for which the Wall Time

metric exceeded 5 minutes. As can be seen, most time was spent in the `groupByKey` transform for the 29-day sample which had a wall time of 28 hours, 6 minutes and 27 seconds. As described in Section 2.4, the Wall Time metric is aggregated over all workers and threads, which is why the wall time for a transform can be greater than the total run time of the experiment.

The average throughput of Harpooner 2 during the experiments can be seen in Figure 4.8a. Just as for Harpooner 1, the throughput scaled linearly as the number of keys increased, but here the line tapered off when the number of keys reached 12,000 instead of 200 as was the case with Harpooner 1.



(a) A graph showing the average throughput of Harpooner 2 during the experiments.

(b) A graph showing the average latency of Harpooner 2 during the experiments.

Figure 4.8: The average throughput and latency for Harpooner 2 for five different numbers of keys.

The average latency of Harpooner 2, which can be seen in Figure 4.8b, also followed a pattern that was similar to the latency of Harpooner 1. Just as with Harpooner 1, the latency stayed fairly constant at around 6-8 seconds until the number of keys reached the point where the pipeline was not able to keep up with the rate at which new events were being generated. The maximum number of keys that Harpooner 2 was able to handle with a latency below 10 seconds was 6,000, which was 60 times more than what Harpooner 1 was able to handle.

Pipeline	**Wall time of the stateful `parDo` transform (hh:mm:ss)**
Harpooner 2	2:41:30
Harpooner 3	6:30:30

Table 4.6: The wall times for the stateful `parDo` transform in Harpooner 2 and 3. The wall times for the other transforms in the pipelines were all shorter than 5 minutes.

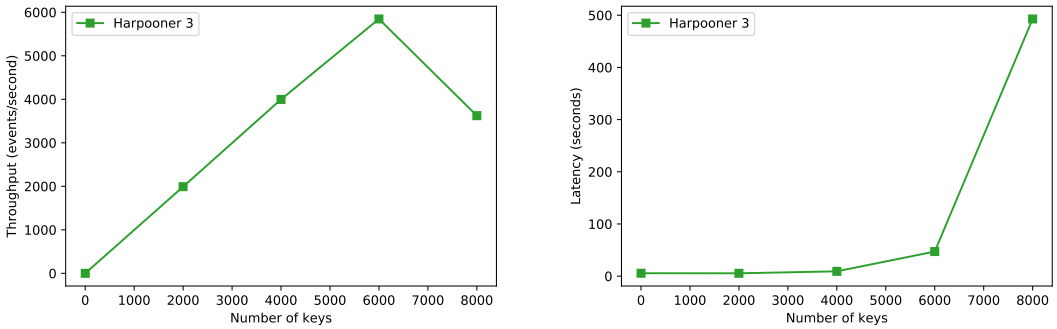
As shown in Table 4.6, the stateful `parDo` transform that applies the K-S test had a wall time of almost 2 hours and 42 minutes in Harpooner 2. All the other transforms

had wall times shorter than 5 minutes. A breakdown of the wall time spent inside this transform, calculated from the profiling data from the profiling runs, is found in Table 4.7. In this table, *Other* refers to all the work done in the stateful `parDo` transform outside of the Insert and Poll operations, which includes the serialisation and deserialisation of state, and *Garbage collection* refers to the time spent on freeing up memory occupied by unused objects. Most of the time in the stateful `parDo` transform in Harpooner 2 was spent on the Poll operation, which had a wall time of 4 minutes and 54 seconds. When analysing the profiling data, we found that the wall time for the Poll operation was dominated by the calculation of the K-S test, and that this calculation was in turn dominated by the sorting of arrays. In total, the sorting of arrays accounted for 70.97 % of the wall time for the stateful `parDo` transform in Harpooner 2.

Pipeline	**Stateful `parDo` transform**			**Garbage collection**
	Insert	**Poll**	**Other**	
Harpooner 2	00:00:00.480	00:04:54	00:00:45	00:02:01
Harpooner 3	00:00:22	00:07:54	02:05:54	00:56:29

Table 4.7: The wall times for Harpooner 2 and 3, taken from the profiling data from the profiling runs. All times are written on the format hh:mm:ss, except for the time for the Insert operation for Harpooner 2 which also includes milliseconds.

Note that the times in Table 4.6 and Table 4.7 can not be directly compared to each other, since the data for the tables were taken from different sources (the Cloud Monitoring API for Table 4.6, which returns the wall time during the entire run; and the profiling data for Table 4.7, which only includes data that was sampled during the 10-minute measurement period).



(a) A graph showing the average throughput of Harpooner 3 during the experiments.

(b) A graph showing the average latency of Harpooner 3 during the experiments.

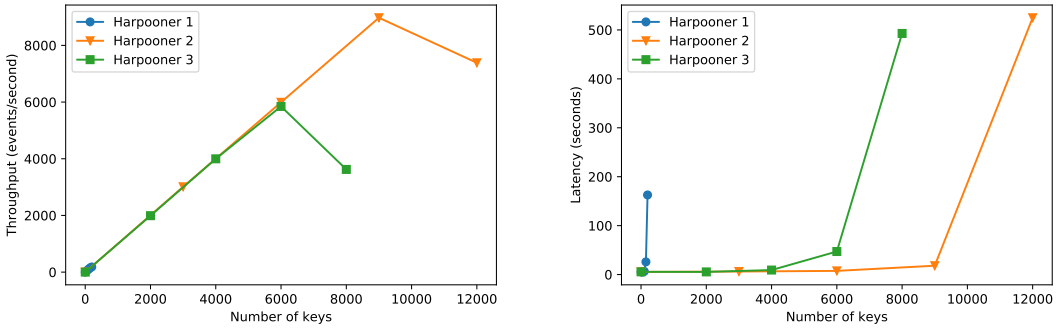
Figure 4.9: The average throughput and latency for Harpooner 3 for five different numbers of keys.

The average throughput and latency of Harpooner 3 can be seen in Figure 4.9a and Figure 4.9b, respectively. These graphs follow the same patterns as the correspond-

ing graphs for Harpooner 1 and 2, but here the performance degraded when the number of keys reached 6,000. The maximum number of keys that Harpooner 3 was able to handle with a latency below 10 seconds was 4,000, which was 40 times more than what Harpooner 1 was able to handle, but only two thirds of what Harpooner 3 was able to handle.

As shown in Table 4.6, the wall time for the stateful `parDo` transform is longer in Harpooner 3 compared to Harpooner 2. Table 4.7 shows that the Poll operation also has a longer wall time in Harpooner 3 compared to in Harpooner 2. When analysing the profiling data, we could see that the wall time for the Poll operation in Harpooner 3 was still dominated by the calculation of the K-S test, which accounted for 87 % of the wall time of this operation. Keeping the BSTs updated only accounted for 3.8 % of the wall time of the Poll operation. This could be an indication that the calculation of the K-S test is actually slower in Harpooner 3 compared to Harpooner 2.

In Table 4.7, we can also see that the wall times for the *Other* and *Garbage collection* entries are longer in Harpooner 3 compared to in Harpooner 2 as well. This could be an indication that the addition of the extra state in Harpooner 3 leads to a lot of extra work for the pipeline, which might be the cause of the decreased performance compared to Harpooner 2.



(a) A graph showing the average throughputs of all three versions of Harpooner during the experiments.

(b) A graph showing the average latencies of all three versions of Harpooner during the experiments.

Figure 4.10: The average throughputs and latencies for the three versions of Harpooner during the experiments.

The average throughput and latency of all three versions of Harpooner is shown in Figure 4.10a and Figure 4.10b, and the maximum number of keys that the pipelines were able to handle with a latency below 10 seconds is shown in Table 4.8. Here, the performance differences between the three versions become apparent, and we can see that Harpooner 2 performed significantly better than both Harpooner 1 and 3.

Pipeline	***k* (number of keys)**	**Average throughput (events/second)**	**Average latency (seconds)**
Harpooner 1	100	100	6.56
Harpooner 2	6,000	5,992	7.52
Harpooner 3	4,000	3,997	9.22

Table 4.8: The maximum number of keys that the anomaly detection parts of Harpooner were able to handle with a latency below 10 seconds.

5

Discussion

In this chapter, we discuss the results from the Evaluation chapter. For the semantics evaluation, we argue that the evaluation method used was good enough to ensure semantic equivalence between Harpooner and an existing batch-processing-based anomaly detection at Spotify. We also discuss the effects of shortening the alert interval from daily to hourly, including why the removal of consecutive alerts was a necessary addition with this change, and whether this removal comes with any negative consequences. For scalability, we discuss the performance of the metric calculation and anomaly detection parts of Harpooner, and what implications the performance of the two separate parts have on the performance of the pipeline as a whole. We also discuss what the bottlenecks were in the three versions of Harpooner, and some future work that could be done to further improve the capabilities of the pipelines.

5.1 Sources of Errors in the Semantics Evaluation

The first research question in this thesis was **Q1** – *How can an anomaly detection system with the same semantics as the existing system, but with an hourly alert interval, be implemented using stream processing?* Here, *existing system* refers to an existing batch-processing-based anomaly detection system at Spotify with a daily alert interval. This question was answered by implementing Harpooner, a stream processing-based counterpart to the existing system with an hourly alert interval.

As stated in **Q1**, the implemented system needed to have the same anomaly detection semantics as the existing system. This was evaluated using data from a single metric at Spotify. In Section 4.2.2 and Section 4.2.3, we argued that this selection of data comprises a sufficient amount of diverse values, but there are two possible sources of errors: (1) only data from a single metric was used, and (2) only data from a finite period of time was used.

As for only using data from a single metric, we argue that this was sufficient because of how the metric was defined and the seasonality it exhibited. The metric used in our evaluation was defined as a ratio between the counts of two event types, and ex-

hibited clear daily, some weekly and possibly some monthly seasonality. Most of the metrics analysed by the existing system are also defined as ratios and exhibit similar seasonality, which means that our metric was representative of these metrics.

As for only using data from a finite period of time, we yet again argue that the selection of data was sufficient, since it ranged over several months and also included several anomalies. As a result, this data captures most of the cases that could appear in data with a longer time period, and hence increasing the time period would not have made a significant difference. Therefore, we consider the chosen test data in total to be sufficient to show that Harpooner has the same anomaly detection semantics as the existing system, and that it can be used to answer research question **Q1**.

5.2 Comparing Daily to Hourly Detection

Our third research question was **Q3** – *How does hourly detection of anomalies compare to daily detection, in terms of which anomalies are detected and how early the anomalies are detected?* Since Harpooner detected anomalies on an hourly basis, while the existing system detected anomalies on a daily basis, the alerts for the two systems can be compared to answer **Q3**. During evaluation, it was shown that Harpooner detected the same anomalies as the existing system, and that these anomalies were detected earlier than or at the same time as the existing system. The earliest Harpooner detected an anomaly during evaluation was 12 hours earlier than the existing system, but theoretically anomalies could have been detected as much as 23 hours earlier with hourly detection compared to with daily detection.

During the evaluation, the exact same anomalies were detected by both Harpooner and the existing system. However, theoretically there could have been cases where Harpooner with its hourly detection would have detected anomalies that could not have been detected with daily detection. For example, in the metric that we used to evaluate the semantics of Harpooner, Segment 3 was anomalous between 2020-02-07 00:00 and 2020-02-07 17:00. If this anomaly would have occurred one hour later, it would still have been detected with hourly detection, but not with daily detection. With this in mind, the question arises regarding how important it is to detect anomalies that only last for a short period of time, and if alerts should even be generated for these types of anomalies. However, if low-latency detection is desirable, it is not possible to wait and see for long an anomaly will last before possibly alerting, which means that there is no reliable way to skip anomalies that only last for shorter periods of time.

5.3 Effects of Removing Consecutive Alerts

In Harpooner, alerts for consecutive hours are removed. This functionality does not exist in the existing system, but was deemed to be a necessary addition when the alert interval was changed from daily to hourly. Without the removal of consecutive alerts during the semantics evaluation, there would have been 183 alerts in total.

With the removal of consecutive alerts, this number was reduced to 3 alerts, one for each anomaly in the metric. Fewer alerts being generated by the system means fewer alerts to investigate, and hence the removal of duplicates leads to less manual work.

However, there are also some disadvantages of removing consecutive alerts. Firstly, with the removal of consecutive alerts there is no way of knowing when a metric is considered normal again after being considered anomalous. Without the removal of consecutive alerts, the absence of an alert means that a metric is considered normal. This is not the case when consecutive alerts are removed, as the absence of an alert could either mean that the metric is considered normal, or that it has been and still is being considered to be anomalous. Secondly, there could also be the case that two alerts for two different anomalies are wrongly considered to be duplicates. For example, assume that the normal behaviour of a metric changes abruptly twice in a short period of time. In this case, the desired behaviour would be to alert twice: one time for each change. However, if the time between the two changes is short enough, the alert for the second change would be considered a duplicate of the alert for the first change, and no new alert would be generated. Both of these cases are rare, and thus, our opinion is that the advantages of removing consecutive alerts outweigh the disadvantages. Also, it would be possible to deal with these cases by adding additional logic to the pipeline in order to alert when a metric is considered normal again, and to re-send an alert when a metric has been considered anomalous for longer than a specified time period.

5.4 Scaling to Handle Spotify's Use Case

When evaluating the scalability of Harpooner, we considered the maximum allowed latency to be 10 seconds. For some areas of anomaly detection, a latency in the seconds could be considered to be too high. For example, anomaly detection is used in autonomous driving [6], an area where a latency close to real-time is needed in order to make the right decisions [38]. However, since our system only detects anomalies once an hour, we consider this latency to be low enough for Spotify's use case.

We used synthetic data when evaluating the scalability of Harpooner, and this data was generated so that data for 1 hour of event time was fed to the pipelines every second during evaluation. Two parameters were used when generating this data: event rate (i.e. the number of events being generated for each hour of event time) and number of keys (i.e. the number of segments in the data).

With a requirement of having a latency below 10 seconds, the maximum average throughput that could be achieved by the metric calculation part of Harpooner was 22,250 events/second. This was achieved with an event rate of 100 events/hour and 113 keys. The maximum event rate that the pipeline was tested on was 1,000 events/hour, where only 5 keys could be handled with a latency below 10 seconds. 1,000 events/hour is low in the context of large scale software companies, which sometimes need to handle hundreds of thousands of events per second [23]. However,

the experiments that generated these results were run using only a single node, while Apache Beam and Cloud Dataflow allows for parallel processing of streams across multiple nodes. Our preliminary results show that if the number of nodes would have been increased, the metric calculation part would have been able to scale to handle Spotify's load.

When evaluating the scalability of the anomaly detection parts of Harpooner, it was shown that different loads could be handled by the different versions. With a requirement of having a latency below 10 seconds, Harpooner 1 could handle 100 keys with an average throughput of 100 events/second, Harpooner 2 could handle 6,000 keys with an average throughput of 5,992 events/second and Harpooner 3 could handle 4,000 keys with an average throughput of 3,997 events/second. Notably, the maximum achievable throughput for the anomaly detection part was lower than for the metric calculation part. However, it is important to recall that the input to the anomaly detection part is only one metric value per hour per key in the data, and not the raw events that are used to calculate the value of the metric. For this reason, the throughput that needs to be handled by the anomaly detection part is only dependent on the number of keys in the data, and not on the event rate. This means that in a real setting the throughput that needs to be handled by the anomaly detection part is a lot lower than for the metric calculation part. Also, just as for the metric calculation part, the experiments that generated these results were run using only a single node. Our preliminary results show that if the number of nodes would have been increased, the anomaly detection part would have been able to scale to handle Spotify's load as well.

During the experiments, it was shown that the anomaly detection part was able to handle a higher number of keys than the metric calculation part. This means that the number of keys that can be handled by the entire Harpooner pipeline would be limited by the number of keys that can be handled by the metric calculation part. For this reason, it might make sense to deploy the two parts as individual pipelines that can be scaled individually. This would also open up for an architecture where multiple metrics are monitored by a single anomaly detection part. As the metric calculation part explicitly takes two streams of raw events and calculates the value of the metric by calculating their ratio, one instance of the metric calculation part can only calculate the value for a single metric. Therefore, one metric calculation pipeline per metric would need to be deployed. However, if the `key` attribute in the output from these metric calculation pipelines is set up to denote a metric-segment combination instead of only a segment, multiple metrics could be monitored by a single anomaly detection pipeline.

5.5 Bottlenecks in the Pipelines

In Harpooner 1, the `groupByKey` transform for the 29-day sample had by far the highest wall time, and the corresponding transform for the 24-hour sample had a high wall time as well. This indicates that these transforms were the bottlenecks of the pipeline. This was expected, because as described in Section 3.3, the sliding windows

for the two samples produced overlapping copies of data, which subsequently needed to be handled by the `groupByKey` transforms.

In Harpooner 2, the sliding windows for the 24-hour and 29-day samples were replaced with a stateful `parDo` transform that keeps a single copy of the data in internal state. This state consisted of a circular buffer and a min-heap. Harpooner 2 was able to handle 60 times as many keys as Harpooner 1, showing the effectiveness of only storing a single copy of the data. The bottleneck of Harpooner 2 was shown to be the computation of the K-S test, for which the complexity was dominated by the sorting of the arrays for the two samples.

In Harpooner 3, the complexity of the K-S test was lowered by introducing two Binary Search Trees (BSTs) from which the K-S statistic was calculated. However, this did not lead to the pipeline being able to handle more keys. On the contrary, Harpooner 3 performed worse than Harpooner 2. We can think of two explanations for this: (1) the sorting of an array is faster than a traversal of a BST due to better cache locality and (2) the resources needed for handling the extra state exceeded the resources saved by reducing the complexity of the test.

As described in Section 3.3, the processing of an event in Harpooner 2 and 3 can be divided into two operations: Insert, which inserts an event into the state; and Poll, which polls the state for a K-S test result. As shown in Table 4.7 in the Evaluation chapter, the wall time for the Poll operation was longer in Harpooner 3 than in Harpooner 2. When analysing the profiling data, we could also see that the Poll operation in Harpooner 3 was dominated by the calculation of the K-S test, and that the updating of the BSTs only accounted for 3.7 % of the wall time. Taken together, these two facts indicates that the calculation of the K-S test was actually slower in Harpooner 3 than in Harpooner 2, even though the complexity of the calculation was lower in Harpooner 3.

As mentioned above, the complexity of the calculation of the K-S test in Harpooner 2 was dominated by calculation of the K-S statistic, and the complexity of this calculation was in turn dominated by the sorting of the arrays for the two samples. This sorting is done using a Dual-Pivot QuickSort algorithm with complexity $\mathcal{O}(n \log(n))$. In Harpooner 3, the K-S statistic is instead calculated by in-order traversal of two BSTs, with the complexity $\mathcal{O}(n)$ – a complexity that is lower than in Harpooner 2. Judging only from the complexity of the two implementations, the calculation of the K-S test in Harpooner 3 should be faster than in Harpooner 2. However, it is important to keep in mind that complexity in itself is not a measurement of how fast an algorithm is, but rather of how well an algorithm scales as the size of the input increases. In Harpooner, the samples for the K-S test have a combined size of 720 elements, which might even be low enough for all the values to fit in memory. QuickSort is considered by many to be the fastest comparison-based sorting algorithm when the data fits in memory [25], so it is quite possible that the sorting of the arrays is faster in practice than the traversal of the BSTs. However, since we view the profiling feature of Cloud Dataflow as experimental, we do not want to draw any definitive conclusions from this data. To determine which of the

two methods of calculation of the K-S statistic is the fastest in practice, we would need a dedicated benchmark that evaluates the two methods in isolation.

To explore the resources needed for handling the extra state introduced with the addition of the two BSTs in Harpooner 3, we can once again look at Table 4.7 in the Evaulation chapter. Here, we can see that the wall times for the *Other* and *Garbage collection* entries were significantly longer in Harpooner 3 compared to Harpooner 2. *Other* in this case includes all the work done in the stateful `parDo` transform outside of the Insert and Poll operations, which includes the serialisation and deserialisation of state, and *Garbage collection* is the time spent on freeing up memory occupied by unused objects. The increase in wall times for these two entries indicates that the addition of the extra state in Harpooner 3 leads to a lot of extra work for the pipeline, which is likely the cause of the decreased performance compared to Harpooner 2.

5.6 Future Work

For future work, we believe that there are some aspects in which the scalability of Harpooner could be further improved. In both Harpooner 2 and 3, the Insert operation could have been optimised by not inserting events into the min-heap if they can be inserted directly into the circular buffer. However, as shown in Table 4.7 in the Evaluation chapter, which presents a breakdown of the wall time spent in the stateful `parDo` transform in Harpooner 2 and 3, the Insert operation only contributed to a small part of the total wall time spent in the `parDo` transform. Therefore, adding this optimisation to the Insert operation might not lead to any noticeable improvements in scalability, but it would make the pipeline more efficient nonetheless.

Furthermore, the performance of both Harpooner 2 and 3 can potentially be improved by making changes to their states. For the serialisation and deserialising of state we used standard Java serialisation, which is comparatively slow [40]. Therefore, using a different serialisation library could be a simple way to increase the performance of the pipelines. Another way to increase the performance could be to reduce the amount of data that is stored in the state. In Harpooner 3, the `java.util.TreeSet` collection was used for the BSTs. This collection does not not allow duplicates to be stored, which was why we included the timestamp of the events in addition to the metric values when storing the events in the BSTs. Implementing a custom BST which allows duplicate values to be stored would remove the need for storing timestamps in the BSTs, which would decrease the total size of the state and possibly increase the performance of the pipeline.

Another aspect that would be interesting to research is the interval at which anomalies are detected. A requirement of Harpooner was to retain the same anomaly detection semantics as the existing system. In the existing system, the metrics were calculated with an hourly granularity, which meant that the shortest possible detection interval was hourly. To achieve an even lower latency, it could be interesting to see how the system would handle a decreased metric calculation interval, such as

calculating the value of the metric each minute. This would open up for a minute by minute detection of anomalies, but it would also come with some additional challenges as the anomaly detection part would have to process 60 times as many values.

To keep the same anomaly detection semantics as the existing system, it was important to use the same computation of the K-S test. However, if this requirement would have been relaxed, approximate K-S tests that are streaming-adapted could be investigated, such as using quantile sketches [24] or computing an estimate of the K-S statistic [29]. These tests trade performance benefits against exactness in the outcome, and are described in further detail in the Related Work chapter.

If the requirement of the anomaly detection were removed, a complete change of anomaly detection model could have been investigated. In this thesis, the K-S test was the only model used for anomaly detection and one challenge was to apply it efficiently in a streaming context. Even if our implementation managed to improve how the samples were stored, the original K-S test is not optimal for streaming systems, as it always compares two entire samples against each other in batches. Additionally, the K-S test itself does not take time into consideration but considers all points in the samples to be equally important. In our case, this means that the metric values that are one month old are considered equally important as the metric values from one day ago. This, together with the fact that the current implementation is using a 29-day sample, decreased its ability to adapt to new normal behaviours that often arises in metrics. Thus, one should consider the possibility to change algorithm entirely. Other algorithms that have been used for anomaly detection in streaming data include Hierarchical Temporal Memory (HTM) [3] and Prophet [34]. HTM is an online sequence memory algorithm that has been proven to work well with a variety of data streams and in cases when time is an important factor. Prophet on the other hand, is an additive model which have been shown to be able to quickly adapt to new normal behaviour, and works well when there are multiple cases of seasonality, such as both daily and weekly. The implementation of one of these algorithms could possibly improve the performance and the anomaly detection capabilities of Harpooner. However, since Harpooner was developed with only the K-S test in mind, some parts of the pipeline would likely need to be redesigned if the algorithm is changed, such as the filtering of p-values and the removal of consecutive alerts.

6

Related Work

In this thesis, we used stream processing to detect anomalies in metrics at Spotify, a large scale software company. Another large scale software company that does anomaly detection using stream processing is Yahoo, which uses the stream processing engine Apache Storm [35] in a generic anomaly detection framework [26]. The focus of this framework is to allow for different anomaly detection algorithms to be used for different data streams. It is recognised that there is no one-size-fits-all solution, and that the best solution in Yahoo's case is to allow different teams to tailor the algorithms used for their specific needs. In our thesis, the focus has rather been on a single algorithm, the Kolmogorov-Smirnov (K-S) test, and how it can be applied efficiently using stream processing. Focusing on a single algorithm allowed us to develop a system that could apply this algorithm in an efficient manner, which would have been harder to do in a more generic system designed to allow different algorithms to be swapped in at will.

The K-S test is inherently designed for use with batch processing, as it compares two samples to each other, but there have been some research on adapting it for analysing data streams. One approach is to reduce the amount of data that needs to be stored when processing the stream by using a *quantile sketch* [24]. A quantile sketch is a data structure that can be used to summarize a set of values, while still retaining approximately the same distribution of values as the original set [18]. In the previously mentioned approach, a quantile sketch is used to summarize the stream of values as it arrives, and the K-S test can then be calculated from this summary with a bounded error. Another approach is to split the stream of values into smaller chunks, and compute an average K-S statistic over the chunks as they are processed [29]. Both of these approaches provide approximative solutions to the K-S test, meaning that they trade speed of computation for exactness in the results. In our case, a requirement was to retain the semantics of an existing batch-processing-based anomaly detection system at Spotify, which ruled out approaches yielding approximative solutions.

Another interesting approach is presented by Kifer et al [21]. In this approach, a data structure called a *K-S structure* is used to allow for fast recomputation of the K-S statistic on data streams. This structure is defined as a balanced tree, which stores the values of the two samples used in the K-S test in its leaves. The values are stored together with a weight, and the nodes in the tree carries information

about their descending leaves. Using this extra information, it is shown how the K-S statistic can be recomputed in logarithmic time. In Harpooner 3, we opted for recomputing the K-S statistic from scratch every time an event is processed, which is done in linear time. While the complexity of the approach proposed by Kifer et al. is lower than the complexity of our approach, the addition of a weight would further increase the size of the state in the system. As was shown in our evaluation, the handling of state proved to be a bottleneck in our pipeline, so lowering the complexity of the computation of the K-S statistic by increasing the amount of data needed to be stored is unlikely to yield better results.

In Harpooner 2 and 3, a circular buffer is used to simulate a sliding window. This is also done in Time-SWAD, a stream processing engine that supports event-time windowing through the use of a flexible circular buffer with support for bulk evictions [10]. This approach is similar to ours, but because our window always contains a fixed number of events, and only one event is removed from the window at a time, a simpler circular buffer with a fixed size could be used instead of a more complex flexible circular buffer.

7

Conclusion

This chapter concludes this thesis by providing answers to the three research questions. The first question was:

Q1: How can an anomaly detection system with the same semantics as the existing system, but with an hourly alert interval, be implemented using stream processing?

Here, *existing system* refers to a batch-processing-based anomaly detection system at Spotify with a daily alert interval. This question was answered by implementing *Harpooner* – a stream-processing-based anomaly detection system with the same semantics as the existing system, but with an hourly alert interval. This system was implemented using Scio, a Scala API for Apache Beam.

The existing system used the Kolmogorov-Smirnov (K-S) test to detect anomalies in metrics, and since our system was to retain the same semantics as the existing system, we needed to use this test as well. To determine how to apply this test in our system, the second research question was:

Q2: How can the Kolmogorov-Smirnov test be applied efficiently on a data stream when using stream processing?

This question was answered by implementing three versions of Harpooner which weighed trade-offs between implementation simplicity, data storage and computational complexity of the K-S test. These versions were: Harpooner 1, which used the built-in sliding windows in Apache Beam to produce the samples for the K-S test; Harpooner 2, which used a stateful `parDo` transform with custom data structures to reduce the amount of intermediate data needed to be handled by the system; and Harpooner 3, which utilised two binary search trees to reduce the complexity of computing the K-S test.

The Harpooner pipelines consisted of two parts: a metric calculation part, which was identical in all three versions; and an anomaly detection part. During evaluation, the two parts were evaluated separately. Thus, four different pipelines were evaluated: one metric calculation pipeline and three anomaly detection pipelines. It was shown that Harpooner 2 had the most efficient anomaly detection part, and that this part was able to detect anomalies in a metric with 6,000 segments with a latency below 10 seconds when run on a single node on Cloud Dataflow. For use in a real setting,

we discussed the possibility of deploying the two parts as separate pipelines. We also argued that when deployed in this fashion, Harpooner would be able to scale to handle the load necessary to do anomaly detection in metrics at Spotify.

The Harpooner pipelines were implemented with hourly detection, as opposed to the daily detection done by the existing system. In order to evaluate the effects of this shortened alert interval, the third and final research question was:

Q3: How does hourly detection of anomalies compare to daily detection, in terms of which anomalies are detected and how early the anomalies are detected?

This question was answered by comparing the alerts generated by Harpooner to the alerts generated by the existing system when evaluating for a metric at Spotify. It was shown that the same anomalies were found by both systems, and that Harpooner detected the anomalies earlier, or at the same time, as the existing system.

All in all, we found stream processing and Apache Beam to be suitable technologies to use when detecting anomalies in metrics with low latency. Additionally, Harpooner provided valuable insights in how an anomaly detection system using these technologies can be implemented.

Bibliography

[1] JavaDoc for java.util.Arrays (Java Platform SE 7). URL: https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html.

[2] JavaDoc for org.apache.commons.math4.stat.inference.KolmogorovSmirnovTest (Apache Commons Math 4.0-SNAPSHOT). URL: http://commons.apache.org/proper/commons-math/apidocs/org/apache/commons/math4/stat/inference/KolmogorovSmirnovTest.html.

[3] Subutai Ahmad, Alexander Lavin, Scott Purdy, and Zuha Agha. Unsupervised Real-Time Anomaly Detection for Streaming Data. *Neurocomputing*, 262:134–147, 11 2017. doi:10.1016/j.neucom.2017.04.070.

[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment*, 8:1792–1803, 2015. doi:10.14778/2824032.2824076.

[5] Tyler Akidau, Slava Chernyak, and Reuven Lax. *Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing.* O'Reilly Media, Inc., 2018.

[6] Riyaz Ahamed Ariyaluran Habeeb, Fariza Nasaruddin, Abdullah Gani, Ibrahim Abaker Targio Hashem, Ejaz Ahmed, and Muhammad Imran. Real-Time Big Data Processing for Anomaly Detection: A Survey. *International Journal of Information Management*, 45:289–307, 4 2019. doi:10.1016/j.ijinfomgt.2018.08.006.

[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[8] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly Detection: A Survey. *ACM Computing Surveys*, 41(3), 2009.

[9] Prajith Ramakrishnan Geethakumari, Vincenzo Gulisano, Bo Joel Svensson,

Pedro Trancoso, and Ioannis Sourdis. Single Window Stream Aggregation Using Reconfigurable Hardware. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 112–119, 2017. doi:10.1109/FPT.2017.8280128.

[10] Prajith Ramakrishnan Geethakumari, Vincenzo Gulisano, Pedro Trancoso, and Ioannis Sourdis. Time-SWAD: A Dataflow Engine for Time-Based Single Window Stream Aggregation. In *2019 International Conference on Field Programmable Technology (ICFPT)*, pages 72–80, 2019. doi:10.1109/ICFPT47387.2019.00017.

[11] Google. Cloud Monitoring API. URL: https://cloud.google.com/monitoring/api/ref_v3/rest.

[12] Google. Cloud Pub/Sub. URL: https://cloud.google.com/pubsub.

[13] Google. Compute Engine Machine Types. URL: https://cloud.google.com/compute/docs/machine-types.

[14] Google. Dataflow: Stream & Batch Processing. URL: https://cloud.google.com/dataflow.

[15] Google. Google Cloud Computing Services. URL: https://cloud.google.com/.

[16] Google. pprof. URL: https://opensource.google/projects/pprof.

[17] Google. Using the Dataflow Monitoring Interface. URL: https://cloud.google.com/dataflow/docs/guides/using-monitoring-intf.

[18] Michael Greenwald and Sanjeev Khanna. Space-Efficient Online Computation of Quantile Summaries. *ACM SIGMOD Record*, 30(2):58–65, 6 2001. doi:10.1145/376284.375670.

[19] Vincenzo Gulisano, Magnus Almgren, and Marina Papatriantafilou. METIS: A Two-Tier Intrusion Detection System for Advanced Metering Infrastructures. In *International Conference on Security and Privacy in Communication Networks. SecureComm 2014. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, volume 153, pages 51–68. Springer Verlag, 2015. doi:10.1007/978-3-319-23802-9{_}7.

[20] Niklas Gustavsson. Views from the Cloud, 2019. URL: https://labs.spotify.com/2019/12/09/views-from-the-cloud-a-history-of-spotifys-journey-to-the-cloud-part-1/.

[21] D Kifer, S Bendavid, and J Gehrke. Detecting Change in Data Streams. In *30th VLDB Conference*, pages 180–191, 2004. doi:10.1016/b978-012088469-8/50019-x.

[22] Kenneth Knowles. Stateful Processing with Apache Beam, 2017. URL: https:

//beam.apache.org/blog/stateful-processing/.

[23] Raffi Krikorian. New Tweets Per Second Record, and How!, 2013. URL: https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.html.

[24] Ashwin Lall. Data Streaming Algorithms for the Kolmogorov-Smirnov Test. In *2015 IEEE International Conference on Big Data (Big Data)*. Institute of Electrical and Electronics Engineers Inc., 12 2015. doi:10.1109/BigData.2015.7363746.

[25] Anthony LaMarca and Richard E Ladner. The Influence of Caches on the Performance of Sorting. *Journal of Algorithms*, 31(1):66–104, 1999. doi:10.1006/jagm.1998.0985.

[26] Nikolay Laptev, Saeed Amizadeh, and Ian Flint. Generic and Scalable Framework for Automated Time-series Anomaly Detection. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1939–1947, New York, New York, USA, 8 2015. Association for Computing Machinery. doi:10.1145/2783258.2788611.

[27] Neville Li. Big Data Processing at Spotify: The Road to Scio (Part 1), 2017. URL: https://labs.spotify.com/2017/10/16/big-data-processing-at-spotify-the-road-to-scio-part-1/.

[28] Frank J Massey Jr. The Kolmogorov-Smirnov Test for Goodness of Fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951. doi:10.1080/01621459.1951.10500769.

[29] Hien Duy Nguyen. A Stream-Suitable Kolmogorov-Smirnov-Type Test for Big Data Analysis. *arXiv preprint*, 2017.

[30] Arnon. Rotem-Gal-Oz. Fallacies of Distributed Computing Explained. 2006.

[31] Richard Simard and Pierre L'Ecuyer. Computing the two-sided Kolmogorov-Smirnov distribution. *Journal of Statistical Software*, 39(11):1–18, 3 2011. doi:10.18637/jss.v039.i11.

[32] Nikolai Smirnov. Table for Estimating the Goodness of Fit of Empirical Distributions. *The Annals of Mathematical Statistics*, 19(2):279–281, 1948.

[33] Spotify. Spotify Technology S.A. Announces Financial Results for First Quarter 2020, 2020. URL: https://investors.spotify.com/financials/press-release-details/2020/Spotify-Technology-SA-Announces-Financial-Results-for-First-Quarter-2020/default.aspx.

[34] Sean J Taylor and Benjamin Letham. Forecasting at Scale. *PeerJ Preprints 5:e3190v2*, 2017. doi:10.7287/peerj.preprints.3190v2.

[35] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm @Twitter. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 147–156. Association for Computing Machinery, 2014. doi:10.1145/2588555.2595641.

[36] Joris van Rooij, Vincenzo Gulisano, and Marina Papatriantafilou. LoCoVolt: Distributed Detection of Broken Meters in Smart Grids Through Stream Processing. In *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems*, volume 18, pages 171–182, New York, NY, USA, 6 2018. ACM. doi:10.1145/3210284.3210298.

[37] Joris van Rooij, Johan Swetzén, Vincenzo Gulisano, Magnus Almgren, and Marina Papatriantafilou. EChIDNA: Continuous Data Validation in Advanced Metering Infrastructures. In *2018 IEEE International Energy Conference (ENERGYCON)*, pages 1–6. Institute of Electrical and Electronics Engineers Inc., 6 2018. doi:10.1109/ENERGYCON.2018.8398800.

[38] Bichen Wu, Forrest Iandola, Peter H Jin, and Kurt Keutzer. SqueezeDet: Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection for Autonomous Driving. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 446–454, 2017. doi:10.1109/CVPRW.2017.60.

[39] Haowen Xu, Wenxiao Chen, Nengwen Zhao, Zeyan Li, Jiahao Bu, Zhihan Li, Ying Liu, Youjian Zhao, Dan Pei, Yang Feng, Jie Chen, Zhaogang Wang, and Honglin Qiao. Unsupervised Anomaly Detection via Variational Auto-Encoder for Seasonal KPIs in Web Applications. In *Proceedings of the 2018 World Wide Web Conference*, pages 187–196. ACM, 2018. doi:10.1145/3178876.3185996.

[40] Yao Zhao, Fei Hu, and Haopeng Chen. An Adaptive Tuning Strategy on Spark Based on In-memory Computation Characteristics. In *18th International Conference on Advanced Communication Technology, ICACT*, pages 484–488, 2016. doi:10.1109/ICACT.2016.7423442.