FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Lightweight Real-Time Data Monitoring

João Dias Conde Azevedo

PREPARAÇÃO DA DISSERTAÇÃO

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. André Monteiro de Oliveira Restivo

Second Supervisor: Prof. Pedro Manuel Pinto Ribeiro

Company Supervisor: Eng. Pedro Cardoso Silva

February 10, 2020

Lightweight Real-Time Data Monitoring

João Dias Conde Azevedo

Mestrado Integrado em Engenharia Informática e Computação

February 10, 2020

Abstract

Data pattern shift detection has been researched within diverse areas and application domains and refers to the process of finding patterns in data that do not conform to expected or usual behavior. Accurate and timely detection of data pattern deviations allows for immediate measures to be taken, preventing loss of income and maintaining the companies' reputation among customers. As such, many large companies take this into consideration and deploy data anomaly components in their systems. In this way, system administrators can proactively adjust and prevent potential service failures.

For example, Yahoo's anomaly detection system fulfills its purpose of monitoring and raising alerts on time-series events for several of Yahoo's use cases. At Microsoft, their custom, internally made anomaly detection system monitors millions of metrics coming from a variety of different internal services, enabling engineers to solve live on-site issues before they scale and become bigger ones.

Many real-time stream monitoring systems are static once deployed in a production environment. Engineers of said systems usually configure them under the assumption that future data flowing through the system follows roughly the same distribution as previously seen data. They do so consciously, to the best of their knowledge and available tools, keeping in mind that in the future they may need to reconfigure the system. Thus, even though the initial configuration of the system may be one of the best fits, over time, due to data pattern shifts, the initially deployed static system's performance gradually deteriorates.

With this thesis, we set out to design and evaluate a system that makes use of lightweight streaming analytical methods to detect deviations in data patterns while in the presence of large volumes of high velocity, highly skewed, seasonal data.

Keywords: real-time, streaming, lightweight, monitoring, volume, velocity, variety, seasonal

Contents

List of Figures

List of Tables

Abbreviations

CMS	Count-Min Sketch
HLL	HyperLogLog
SPE	Stream Processing Engine
SWAG	Sliding Window Aggregation

Chapter 1

Introduction

1.1 Context

In the past decade, applications have become increasingly data-driven, placing data at the center of application design. Different use cases process data in different ways for different purposes. For instance, e-commerce platforms need to process a large number of transactions, for example, those made daily by all Alibaba's clients while ensuring that sales run smoothly and that products are delivered to customers' homes. Streaming services, like YouTube and Netflix, try to guarantee that media content reaches up to millions of users simultaneously. Furthermore, this media content may have been previously recorded and stored, or may be generated on the fly. Social networks are also responsible for generating large volumes of data. A social network like Twitter, for example, generates more than 500 million *tweets* (posts in Twitter) per day [28]. *Tweets* may contain text, media content or both. Cybersecurity applications are yet another example of time-sensitive, data-driven applications. These need to monitor user accesses and their actions in the network, where timely detection of intruders is critical to prevent them from tampering with the protected system.

All of these applications generate large volumes of data over time which poses data storage and processing challenges. This data is characterized in terms of its large volume, high velocity and high variety [23], creating large scale data management problems. Additionally, the large amounts of information produced by such systems and use-cases lead to the creation of multiple real-time unbounded data sets, also known as data streams. The information that flows through such a stream can be analyzed on-the-fly or stored for later processing. The former case is best

known as stream processing, while the latter as batch processing. E-commerce applications have to process endless streams of credit card transactions. Live streaming services need to distribute the media content created by the *streamer* (content creator) and deliver it across multiple users. Social networks process and analyze hundreds of posts per second to find out trending keywords across their network and suggest them to users. Computer network monitor systems want to detect intruders and revoke them system access before they have time to corrupt the system or steal private information. All of the previously given use cases rely on real-time processing of data for near real-time actions. Hence, between batch and stream processing, the latter fits these use cases better.

As mentioned, the processing of data can be divided into two main categories: batch and stream processing. However, due to the increasing volume of data and the need for timely processing to allow companies to react to changing conditions in real-time, stream processing demand is increasing. Data stream processing refers to the handling of huge volumes of high variety of data generated at high-velocity from multiple sources in real-time. Unlike batch processing on static data sets, where assumptions on the underlying data distribution can be made, streaming data is known for being non-stationary [14]. Furthermore, in the streaming paradigm, the value of the produced information lies in its recency [21]. *Recency* is measured under different scales for different use cases. For instance, considering the monitoring of geological data to forecast possible natural disasters such as earthquakes versus the monitoring of a computer network for intruder detection. In the former case, after the forecast of a future earthquake, governmental authorities need a couple of days to launch an evacuation plan and keep the population safe. However, in the intruder detection scenario, the decision of removing access to a user must be done as soon as possible, preferably in a few seconds. Hence, while in the first scenario information retrieved within a day would still be recent, in the second scenario, information is considered recent and relevant if delivered within seconds (or even milliseconds).

Both in batch and stream processing, we usually want to apply operations on data, for example, to compute the maximum value, an average or a count of distinct elements. Operations applied to data sets that produce a single result are commonly known as aggregate operations. In stream processing, we apply these aggregations to data streams to obtain valuable information. However, since a data stream is, effectively speaking, an unbounded data set, if our aggregation requires a finite domain, there is the need to apply windowing techniques. A thorough analysis to stream windowing techniques is done in the Background Section of this document but for now it suffices to say that windowing transforms an unbounded data stream into a finite data set by defining a time or tuple-based window size — *i.e.* events from the last three days or the last 10^6 events, respectively.

Consider the case is the detection of credit card fraud on the data stream of all the transactions made on Amazon by every US citizen on a single day. In this case, Amazon's transaction stream will need to be monitored by another system, a fraud detection one. Such a system would have to process all the incoming transactions and decide in a fraction of a second if it is a fraudulent one or not. It is clear that manually monitoring such an intensive, latency-sensitive data stream is

hard and error-prone. Thus justifying the need for autonomous real-time data stream monitoring systems.

Many market solutions monitor streams of credit card transactions with the intent of fraud detection. In the context of this Thesis, we will work side by side with Feedzai. Feedzai offers fraud prevention solutions for the previously presented use case of monitoring an infinite stream of credit card transactions and detecting fraud. We will work side by side with them, taking advantage of the multiple real-life data sets from the financial fraud space they can provide.

Feedzai's solution for fraud detection consists of a workflow of different components. Incoming events are fed into this workflow and processed accordingly. Workflow components can be sets of user-defined rules and/or Machine Learning (ML) models that output scores reflecting their belief on whether a transaction is fraudulent or not. ML models are configured and trained before deployment making assumptions on the statistical distributions of future incoming data. Thus their model performance is linked to the maintenance of these assumptions. However, fraud patterns are not static over time, but seasonal — *e.g.* Black Fridays, holiday shopping and product launches — and evolutionary — *e.g.* new fraud attack methods by fraudsters. Due to these shifts in data patterns and distributions, the ML models' performance can deteriorate over periods of time. In the Machine Learning field, the performance decay caused by data pattern shifts is also known as concept drift.

1.2 Objectives

Feedzai's data pattern shift challenge is not unique to them nor their field. Many analytical systems are static once deployed and are configured *a priori* under the assumption that future data flowing through the system will roughly follow the same distribution as previously seen data. The problem is that data pattern shifts will lead to poor performance by the previously configured system. Monitoring the data patterns themselves and alerting for changes in the underlying distribution would allow users to reconfigure the system before its performance declines.

The objective of this Thesis is to build a lightweight real-time system that detects these shifts in data patterns in systems that handle high volumes of high velocity, highly skewed and seasonal data.

1.3 Motivation

For any organization, the quality of their provided services is paramount. Therefore, it is common for companies to have systems monitoring the former, testing their usage rate, availability and performance, just to name a few. The motivation behind this Thesis is to produce a system capable of real-time monitoring a stream of data so that changes in the underlying stream are caught early, preventing a fall in performance.

We want to build a lightweight enough solution that can be integrated into existing workflows. Ensuring constant time and space complexity means that the system will remain usable for the

ever-growing volumes of data. Since such a monitoring system is not mission-critical — *i.e.* the provided services don't rely on it — the operational cost associated with it must be kept to a minimum. Having a real-time system monitoring other real-time systems consuming as many resources as the latter would be far too costly.

1.4 Hypothesis

The question this Thesis aims to answer is whether sliding window aggregations and probabilistic data structures can be used to detect data pattern shifts in data streaming scenarios in the presence of high volumes of high velocity, highly skewed and seasonal data with a low memory footprint.

1.5 Hypothesis Validation

The hypothesis will be proven true or false depending on whether the following criteria are met. The resulting data pattern shift reporting system must have the following properties:

- high precision ratio — *i.e.* at least 80%
- high recall ratio — *i.e.* at least 80%
- low latency — *i.e.* constant time complexity, *O(1)*
- memory efficiency — *i.e.* constant memory complexity, *O(1)*

Experiments will be made using multiple real data sets from the financial fraud space and compared to existing batch analysis results.

1.6 Document structure

This section merely sums up how each of the following Chapters contributes to this Thesis.

In Chapter 2 of this document we brief the reader with the necessary background and relevant concepts needed to understand the following Chapters.

In Chapter 3 we examine the state of the art in the area, analyzing the most recent work done and classifying it into subgroups.

In Chapter 4 we define the scope of this project, the research questions we intend to answer, our experimental methodology and our future planning.

In Chapter 5 we summarize our contributions.

Chapter 2

Background

2.1 Stream processing as a superset of batch processing

Batch processing is the processing of data grouped by batches. A batch is a collection of data points that have been grouped by certain criteria. Examples of such criteria could be *"all the transactions from the past two months"*, *"the last million transactions"* or even *"all the transactions made by an user"*. Batch data sets are known as static data [22]. An example application of such a technique would be processing all the keywords searched for in the google search engine. For instance, grouping the results by keyword and counting how many times a keyword was searched for trend analysis purposes. On the other hand, stream processing refers to the processing of data streams. A data stream is essentially an unbounded data set. This is in contrast to batch processing which works on a finite data set. This distinction is important since some computations are impossible to compute accurately on unbounded data sets. Such an example of an impossible computation on an infinite data set is grouping incoming data by an ID and counting distinct elements. An example of such Stream Processing Engines (SPE) is Apache Flink [3].

Batch processing is considered the oldest between both processing paradigms. Historically speaking, Apache Hadoop [4] was one of the first widely known technologies for large-scale batch processing, created in 2005. Nearly a decade later, Apache Spark [5] was released. Spark is capable of doing not only on-disk processing like Hadoop but also in-memory processing, achieving lower processing times. As more and more data became available, the more need for stream processing increased. In order to provide a streaming solution, Apache Spark created the Spark Streaming module. Being a batch system in its core, Spark implements streaming as micro-batching. This approach handles streaming data by grouping incoming events into very small batches. Then, in order to process the entire defined window of data, it joins all micro-batches

into a larger one and performs the desired computations. However, this creates an artificial barrier that does not really exist in the streaming context. More than conceptually different from a true stream processing system, processing a stream in a batch fashion, even if in small ones, presents some issues. First, since we are indeed working with batches, results will never be real-time updated per event. For example, if we use a batch size of 100 events, the system would only produce accurate results every 100 events. To handle system inactivity usually these micro-batch systems also have the notion of updating the aggregations with incomplete batches, if no event has entered the system after a pre-specified time period has elapsed. As such, micro-batching was created as a natural implementation of stream processing using the available batch processing systems of the time, but can not be considered true stream processing.

Distinguishing between batch and stream processing is important to understand how a monitoring system on top of the latter would work. The Venn diagram 2.1 was taken from [6] and illustrates the relation of both processing techniques. An data stream is unbounded or infinite while a bounded data stream has a beginning and an end. Using these definitions, the content of an unbounded data stream contains that of a bounded data set. That is, from an infinite stream, it is possible to fix a left and right boundary in order to obtain a bounded and finite data set. This relation is visible in the 2.2 diagram from [6].

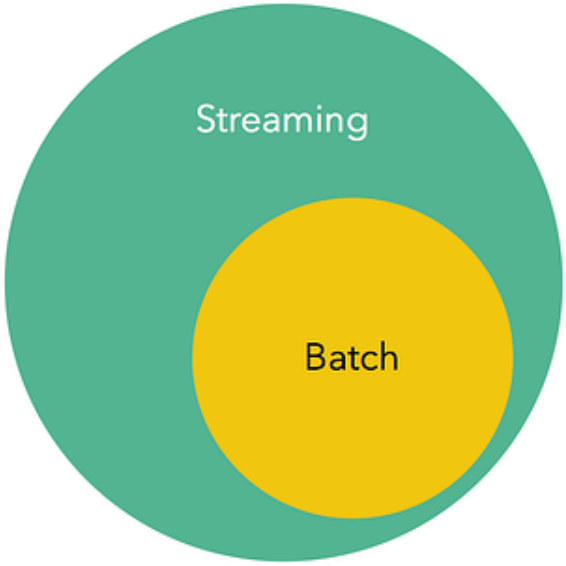

Figure 2.1: Stream processing as a superset of batch processing

In Section 2.2 we will cover exactly how an unbounded data set (also known as a data stream) can be converted into a bounded data set with the use of windows.

2.2 From unbounded streams to finite data sets

In Section 2.1 we have defined a data stream as an unbounded data set. But what exactly are the issues of working with an infinite stream of data? Why might we need to convert it into a bounded

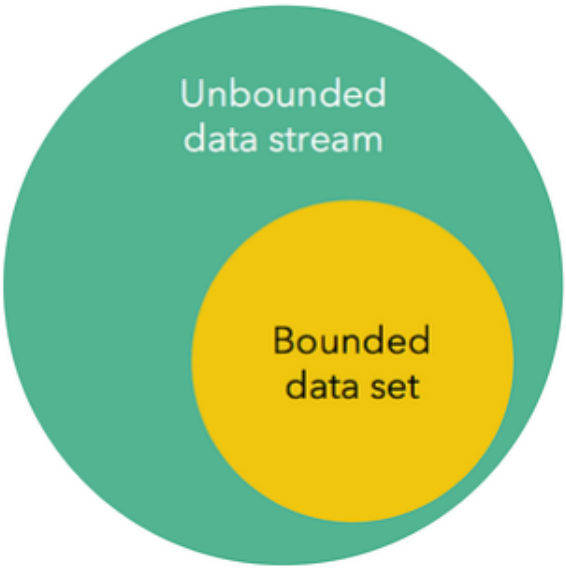

Figure 2.2: Unbounded data streams as a superset of bounded data sets

data set and how to do so? In this Section, we will analyze such issues and present streaming windows as the solution.

Computations over an unbounded data set are not impossible. For example, computing the *average* of an event's field for all incoming events is computationally feasible. However, is this boundless average over the stream of events useful? While in some scenarios it may be, more often than not we would like to introduce some logical context into our computations. For instance, consider the monitoring of machines spread out across multiple racks in a large data center. Assume that temperatures measured across all racks are sent to a monitoring system in the form of a data stream. Is the average of all temperatures of all machines over an infinite period useful for the detection of overheating equipment? In this scenario, an average of the temperatures per rack and for the last 5 minutes allows for actionable context, in this case identifying a likely-to-fail rack. To obtain such insight requires computing the average over the last 5 minutes' worth of data, effectively defining a begin and end. By creating such a conceptual range, we realize that in practice we need a bounded data set. Partitioning an infinite data stream to obtain such a finite data set is made through the use of windows.

The authors of [15] claim that *"recent history is often managed using windows"* and that *"windowing makes it possible to implement streaming versions of the traditionally blocking relational operators, such as streaming aggregations"*. More formally, the authors of [8] state that *"a window W over a stream S is a finite subset of S"*.

Windows can be categorized by two dimensions: the window size W and the sliding step size S, both either tuple or time-based. Tuple-based windows' size is specified by the number of items the window can contain and time-based windows specify the time interval worth of incoming data a window can store. For instance, a tuple-based window of size 10^3 tuples will store exactly 10^3 items. In contrast, a time-based window storing the last 5 minutes of events can contain a varying

number of tuples but will always keep the events arrived in the last 5 minutes. A window slides over incoming data and groups it, enabling us to compute aggregations over a defined and bounded set of events. The step size *S* of a windowing technique is formally defined by [8] as *"the distance between consecutive windows"*. This means that two consecutive windows W_1 and W_2 are exactly *S* units away. As mentioned, these units can be units of time or a number of tuples.

There are three types of windows: true sliding, stepping or tumbling windows. A true sliding window is defined by any given window size *W* and a unitary step (*S*=1), which can be defined as time or tuple-based. At each step, the window advances one unit, which may be one event for tuple-based steps, or one time-unit whether that is in milliseconds, seconds, days, weeks or another magnitude of time, measured based on the system clock or event time. Such a true sliding movement can be observed in Figure 2.3 with a tuple-based window of size *W* of three tuples and unitary step.

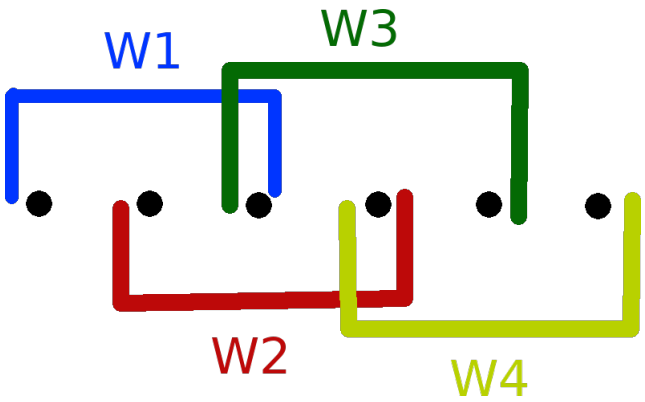

Figure 2.3: True sliding window, W=3, S=1

A stepping window behaves exactly like a true sliding one but the step size *S* is not unitary. In other words, a stepping window is defined by any window size *W* and steps *S* greater than a unit. Once again, both the window and step sizes, *W* and *S* respectively, can be time or tuple-based. Figure 2.4 represents a stepping tuple-based window of size *W* of three and a stepping size *S* of two.

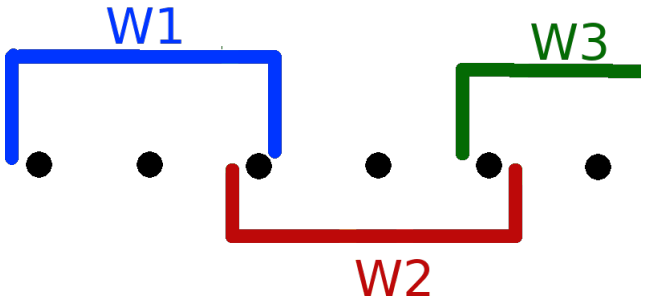

Figure 2.4: Stepping window, W=3, S=2

A tumbling window, also defined by its window size *W* and step size *S*, is a window where W is equal to S. Hence, tumbling windows partition the stream into non-overlapping chunks with the same size. Since the intersection of different tumbling windows is always empty, each event is said to belong to exactly one tumbling window. An example of a tumbling window can be seen in Figure 2.5 with a tuple-based window of size *W* of three and an equally large step size *S*.

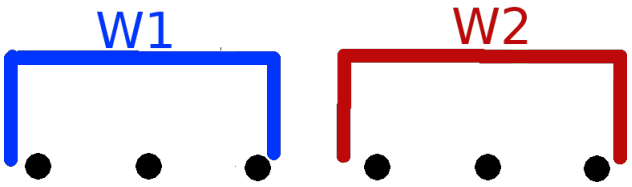

Figure 2.5: Tumbling window, W=S=3

For the scope of this Thesis, we will work under a time-based sliding window framework with a unitary step size. Despite focusing on time-based true sliding windows, our approach is just as valid for tuple sized windows and for other step values as well.

2.3 Aggregations over data streams

Whether working with a static bounded data set or a dynamic data stream bounded by windows, the raw information contained is typically not immediately useful without some computation applied. These computations are known as aggregations. Aggregations over data are fundamental in overall data processing, not just stream processing. There are multiple examples of aggregation operations, such as computing counts, averages, maxima or minima and more. In stream processing, aggregation plays a central role, but unlike many other scenarios that focus on data at rest, streaming aggregations handle data in motion. In the data center temperature monitoring example discussed in Section 2.2, we wanted to group data by rack and aggregate it applying an average aggregation. This illustrates the need for sliding window aggregations.

Some aggregations operate only on a finite data set. As we have seen in Section 2.2, to obtain a bounded data set from a data stream we apply windowing techniques. This way we can apply aggregations to data streams using sliding windows. The authors of [27] define such aggregations over sliding windows as Sliding Window Aggregations (SWAGs).

Once again let us consider the data center temperature monitoring problem from Section 2.2 as an example use-case for SWAGs. How would we compute the average temperature per rack in the data center in the last 5 minutes? To better understand and later on examine this aggregation, let us translate this question to a streaming SQL query, with a similar syntax to the API provided by Apache Flink [3]. Assuming all the temperatures can be queried from the *Temperatures* table and that to each rack corresponds a different *rack_id* we have the following query:

```
SELECT rack_id, AVG(temperature)
FROM Temperatures
```

```
GROUP BY TUMBLE(timestamp, Time.minutes(5), rack_id)
```

In the query above, the *TUMBLE(timestamp, Time.minutes(5), rack_id)* statement groups incoming data using time-based tumbling windows of 5 minutes duration, using the *timestamp* field to filter out events not in this time interval. The resulting list of rows has *rack_id* as its key and a corresponding value a list of temperatures, from the past 5 minutes. Finally, we compute an *AVG* aggregation for each of the racks, resulting in the average temperature per rack in the past 5 minutes.

The distinction between the aggregation algorithm and the aggregation operation is important. For example, the operation we are interested in might be a sum, an average, a maximum, a minimum or something more complex such as a membership test or the count of distinct elements. The aggregation algorithm comes with its own data structures for storing the intermediate aggregation state and a specific algorithm to update this data structure. For example, consider the computation of an aggregation like a sum over data grouped by a true sliding window of any size. One way of computing said sum would be to, after each unitary step, compute from scratch the sum of all window values. However, a more efficient way of doing so would be to compute the total sum of the window once and then, at each step, evict the oldest element and subtract it from the total sum state and include the new event in the window while adding it up to the aggregation state. In this scenario, even though the aggregation operator is the same — *i.e.* a sum — the algorithms differ in that one recalculates everything from scratch and the other incrementally changes the aggregation state. Hence, the operation and the algorithm differ greatly from each other.

Different operations have different properties. These properties are listed in [27], where the authors group different operations according to the properties they satisfy. Table 2.1 was adapted from this work and categorizes different families of operations based on a set of properties.

In order to properly explain these properties it is necessary to first define the *combine* and *lower* functions. Both these functions are first introduced in the work of [27]. The *combine* function merges the incoming event with the incrementally computed aggregation state. The *lower* function is responsible for returning a result for a given query according to the current aggregation state. For instance, consider the previous example of computing the sum over a true sliding window in the past 5 minutes. The *combine* function would be the arithmetic operator + and would merge the current aggregation state — *i.e.* the current sum value — with the new incoming event. The *lower* function would return the current aggregation state — *i.e.* return the total sum computed.

The following are formal definitions of these properties and were summarized from [27]. An aggregation operation is said to have:

1. an ***invertible combine*** if a function $\ominus$ exists such that $(x \oplus y) \ominus y = x$, for all x and y
2. an ***associative combine*** if $(x \oplus y) \oplus z = x \oplus (y \oplus z)$, for all x and y
3. a ***commutative combine*** if $x \oplus y = y \oplus x$, for all x and y
4. a ***size-preserving combine*** if the aggregation state resulting from $\oplus$ always occupies the same space in memory

	invertible `combine`	associative `combine`	commutative `combine`	size-preserving `combine`	unary `lower`
sum-like: sum, count, average, standard deviation, ...	✓	✓	✓	✓	✓
collect-like: collect list, concatenate strings, i^{th}-youngest, ...	✓	✓	×	×	?
median-like: median, percentile, i^{th}-smallest, ...	✓	✓	✓	×	?
max-like: max, min, argMax, argMin, maxCount, ...	×	✓	?	✓	✓
sketch-like: Bloom filter, CountMin,HyperLogLog	×	✓	✓	✓	×

Table 2.1: Aggregation operations and properties. Checkmarks (✓), crosses (×), and question marks (?) indicate a property is true for all, false for all, or false for some of a given group of like operations, respectively

5. a ***unary lower*** if the query does not need any parameters — *e.g.* a membership test with a Bloom filter would require the element to test membership for as a parameter

These properties are important to evaluate SWAG algorithms. Different algorithms might only apply to certain families of aggregations operations. This way, the multiple SWAG algorithms presented in Chapter 3 will be bench-marked not only on their time and space complexity but also on the restrictions they impose regarding the aggregation operator properties.

Chapter 3

State of the Art

In this Chapter, we will discuss the most relevant work for the aforementioned problem of detecting data pattern shifts in real-time computations over true sliding windows using resource-lightweight approaches on streaming systems. We present a wide analysis of several categories of algorithms that seemed like viable options to build the desired system.

3.1 Detecting Change in Data Streams

For static data sets, it is reasonable to assume that the data was generated by a fixed process, for example, the data is a sample from a static distribution. But a data stream has necessarily a temporal dimension, and the underlying process that generates the data stream can change over time [1] [18]. It is this change in the underlying distribution that we want to detect and report as a data pattern shift.

3.1.1 A two window paradigm algorithm for change detection

The authors of [19] propose an algorithm for change detection in data streams. The only assumption made is that data points are generated independently. The authors propose that detecting change in data streams can be reduced to testing if two windows have different underlying distributions. Thus the change detection algorithm works on a two-window paradigm.

Figure 3.1 contains the pseudocode presented by the authors. The proposed method uses two windows, a reference W_1 and a sliding one W_2. The authors used tuple-based windows of size k and the sliding window was a true sliding one, as described in Section 2.2. The reference window W_2 works as a *baseline* and contains the first k points of the stream that occurred after the last detected change.

The algorithm begins by filling both windows with the first k tuples. Then it slides window W_2 (by one, since it is a true sliding window). It does this k times. In between slides, the algorithm checks for change by applying the d function, which measures the discrepancy between the two windows contents, W_1 and W_2. Whenever this discrepancy is above a certain threshold α, the algorithm reports a change in the data stream. Then it repeats the whole process, re-initializing both windows with the next k items and proceeding as already described.

Algorithm 1 : FIND_CHANGE

```
1: for i = 1...k do
2:    c_0 ← 0
3:    Window_{1,i} ← first m_{1,i} points from time c_0
4:    Window_{2,i} ← next m_{2,i} points in stream
5: end for
6: while not at end of stream do
7:    for i = 1...k do
8:       Slide Window_{2,i} by 1 point
9:       if d(Window_{1,i}, Window_{2,i}) > α_i then
10:         c_0 ← current time
11:         Report change at time c_0
12:         Clear all windows and GOTO step 1
13:      end if
14:   end for
15: end while
```

Figure 3.1: Two window method for data stream change detection

This algorithm reduces change detection in data streams to testing whether two samples, the *reference* and the *sliding window* contents, are generated by different distributions. Consequently, the authors study the case of detecting differences in distribution between two samples. This is the purpose of the d function. The authors evaluate many statistical tests as possible implementations of this d function that must truly quantify an intuitive notion of change.

Experiments were made using the following statistical tests as the d function: Wilcoxon signed-rank, ϕ_A and Ξ_A. The Wilcoxon test is well known but the ϕ_A and the Ξ_A statistical tests were defined by the authors. Since computing these three statistical tests for our windows contents takes identical time and space, we consider that a thorough analysis of each one is out of scope for this Thesis, as it does not provide relevant information.

The authors conclude that no test is best in all of the experiments done. However, the ϕ_A and Ξ_A statistics vastly outperformed the others in some tests and did not perform that much worse in other test cases.

Diving deeper in the algorithm presented in 3.1 we now analyze the computation of the d function value. In the authors' work, efficient computation of the previously mentioned statistical tests — *i.e.* the value of function d — is further explored and an algorithm for such is presented. First, the *KS structure* is presented. The formal definition given by the authors is shown in Figure 3.2

Definition 5 (KS structure). We say that A is a KS structure if

- It is a finite array $< a_1, \ldots, a_m >$ of elements in $\mathbb{R}^2$ where the first coordinate is called the "value" and the second coordinate is called the "weight". The value is referred to as $v(a_i)$. The weight is referred to as $w(a_i)$.
- The array is sorted in increasing order by value.
- The length of the array is referred to as $|A|$.

Figure 3.2: Formal definition of the KS structure

The KS structure is built from the reference W_1 and sliding W_2 windows. Consider the pair (W_1, W_2) where the size of both windows is k. Creating the KS structure Z of size $2k$ is done by joining the elements from W_1 and W_2. As stated in 3.2, each element of Z will have a value and a weight. The values will be the actual elements joined. The weights of elements coming from W_1 will be $-1/k$ while the weights of elements coming from W_2 will be $1/k$. The final KS structure Z must be sorted in increasing order by value. The KS structure Z can be maintained throughout running time by making use of a balanced tree, such as a B-tree, in $O(\log(2k))$ time. By making use of the KS structure, all of the mentioned statistics can be computed in $O(2k)$ time.

3.1.1.1 Time complexity analysis

The algorithm begins by collecting the first k tuples. Both windows will have size k. This takes $O(k)$ time but is a task that will only occur on initialization and on change detection, upon which the window contents must reset, as described in 3.1. For each of the next k iterations the function d will be computed and compared to the threshold α. As previously mentioned, computing the value of d can be done in $O(\log(2k))$ time per iteration. However, the initial time cost of initializing the balanced tree is of $O(k \log(2k))$.

Initialization is not frequent. It occurs once at the beginning of execution and once per change detection upon which the algorithm resets the windows' contents. To provide an accurate time complexity, we take into account the probability P of change occurring and being detected leading to a reset. Assuming P is within $[0, 1]$, we have a time complexity of $O(P(k + k \log(2k)))$ per reset — *i.e.* the probability of a reset occurring times $O(k)$ to initialize the windows' contents plus $O(k \log(2k))$ to build the balanced tree.

In runtime, the algorithm will perform k iterations where it computes the d function value, taking $O(\log(2k))$ per iteration, amounting to a time complexity of $O(k \log(2k))$.

Putting both initialization and runtime complexities we have a total time complexity of $O(P(k + k \log(2k)) + k \log(2k))$.

3.1.1.2 Space complexity analysis

The algorithm presented makes use of two k sized windows and one balanced tree of size k as well. This represents a space complexity of $O(k+k+k)$ or $O(3k)$.

3.1.1.3 Applicability to our Hypothesis

The algorithm for change detection presented in Subsection 3.1.1 has time complexity of $O(P(k + k\ log(2k)) + k\ log(2k))$. and space complexity of $O(3k)$, with k as a window size constant.

In this Thesis, we want to detect and alert data pattern shifts so detecting change in data streams is directly applicable to our use case. However, our focus is on a lightweight solution and this algorithm has a linear space complexity, so it does not fit our scope as a possible solution.

3.2 Pattern Mining

Taking on a new approach, in this Section, we explore pattern mining algorithms rather than algorithms that detect change in data streams. The idea behind using pattern mining algorithms being that if we can maintain a list of frequent patterns we would be able to detect when said patterns change — *i.e.* the list changes. Change in the frequent mined patterns would be associated with a data pattern shift and immediately reported.

3.2.1 Mining Frequent Patterns: the FP-Stream structure

The authors of [16] developed a data structure named *FP-Stream* — that summarizes frequent data patterns – and algorithms to build and incrementally maintain it.

We agree with the author's statement that it is *"unrealistic to hold all streaming data in the limited main memory"*. With this in mind, they divide patterns into three categories: frequent, sub-frequent and infrequent patterns. The focus of their work is the mining and maintenance of frequent and sub-frequent patterns since the latter might become frequent later. Thus, infrequent patterns are discarded, using less memory.

The FP-Stream structure consists of a frequent pattern tree (FP-Tree [17]) with tilted-time windows in each of the nodes.

The authors are not very explicit on the definition of a tilted-time window. They claim we are *"often interested in recent changes at a fine granularity, but long term changes at a coarse granularity"* and that tilted-time windows fit such use case. Figure 3.3 shows such a window and was withdrawn from the paper. In it, we see the 4 quarters of the last hour, then the last 24 hours and finally 31 days. They claim that *"one can compute frequent itemsets in the last hour with the precision of quarter of an hour, the last day with the precision of hour, and so on, until the whole month"*.

An FP-Tree [17] as shown in Figure 3.4, is a tree representation of frequent patterns. Each node in the frequent pattern tree represents a pattern and its frequency (*support* column in the

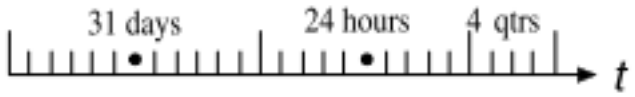

Figure 3.3: Tilted-time window

Figure), recorded in the node. The same tree construction algorithm from [17] is used in the FP-Stream algorithm.

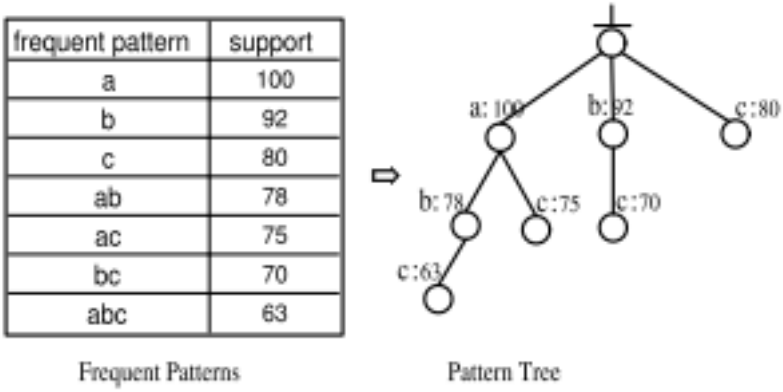

Figure 3.4: FP-Tree

The authors propose using only one frequent pattern tree, where at each node, the frequency for each tilted-time window is maintained. Figure 3.5 shows the FP-Stream structure, as an example of a frequent pattern tree with tilted-time windows embedded in each node.

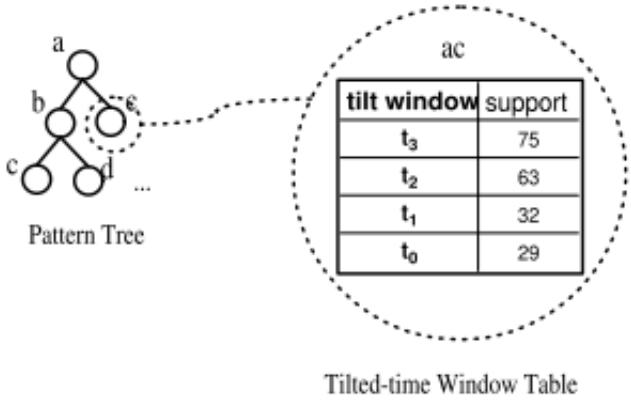

Figure 3.5: FP-Stream

The algorithm for constructing and maintaining the FP-Stream structure is given as a high-level

list of instructions. The algorithm groups incoming streaming data into batches. Initialization is done only when the first batch is complete. As the transactions for the first batch arrived, the frequencies for all items are computed. Once all transactions for the first batch have arrived, the batch is scanned to create an FP-tree structure, pruning all infrequent items – *i.e.* below a certain frequency threshold. Finally, an FP-Stream structure is created by mining the frequent items from the FP-Tree.

The incremental update of the FP-Stream as data arrives is described on a very high level and does not add relevant information to understand time and space complexity without knowledge of the FP-Tree algorithm, on which it heavily relies. Further research will be done to perform a more thorough analysis of this algorithm, starting with the analysis of [17].

3.2.1.1 Applicability to our Hypothesis

No definitive conclusions can be made without the full time and space complexity analysis. However, the maintenance of a frequent pattern tree and tilted-time windows for each of the nodes might reveal to memory intensive, discarding this as a valid solution.

3.2.2 Neighbor-Based Pattern Detection for Windows Over Streaming Data: the Extra-N algorithm

The authors of [29] propose a method for incremental detection of neighbor based patterns, namely density-based clusters and distance-based outliers, specifically for sliding windows scenarios. Figure 3.6 is an example of the previous two mentioned neighbor based patterns. Formal definitions for both distance-based outliers 3.7 and density-based clusters 3.8 are provided.

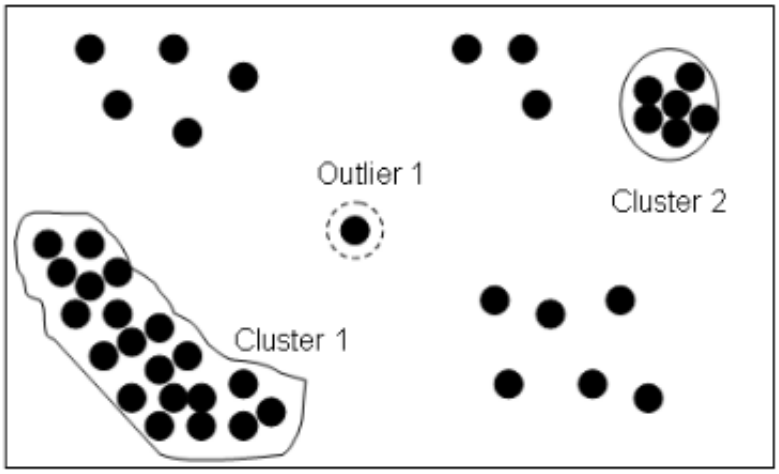

Figure 3.6: Two density-based clusters and one distance-based outlier determined by neighbour-based pattern detection algorithms

Density-based clusters label data points as *core points*, *edge points* or *noise*. These classifications are explained in 3.8. Data points are classified based on the number of neighbors they

Definition 2.1. ***Distance-Based Outlier:*** *Given θ^{range} and a fraction threshold θ^{fra} ($0 \leq \theta^{fra} \leq 1$), a distance-based outlier is a data point p_i, where $NumNei(p_i, \theta^{range}) < N * \theta^{fra}$, with N the number of data points in the data set.*

Figure 3.7: Formal definition of distance-based outliers

Definition 2.2. ***Density-Based Cluster:*** *Given θ^{range} and a count threshold θ^{count}, a data point p_i with $NumNei(p_i, \theta^{range}) \geq \theta^{count}$ is defined as a core point. Otherwise, if p_i is a neighbor of any core object, p_i is an edge object. p_i is a noise if it is neither a core point nor an edge point. Two core points c_0 and c_n are connected, if they are neighbors of each other, or there exists a sequence of core points $c_0, c_1, ...c_{n-1}, c_n$, where for any i with $0 \leq i \leq n-1$, a pair of core objects c_i and c_{i+1} are neighbors of each other. Finally, a density-based cluster is defined as a group of "connected core objects" and the edge objects attached to them. Any pair of core points in a cluster is "connected" with each other.*

Figure 3.8: Formal definition of density-based clusters

have within a range θ^{range}, measured versus a θ^{count}. In brief, if a data point has more than θ^{count} neighbors within θ^{range} it is classified as a *core point*. If it does not but it is a neighbor of a core point, then it is an *edge point*. Otherwise, it is considered *noise*.

The authors make use of the *"predictability"* property of the expiration of existing objects. In other words, given a window with a fixed slide size, they predetermine the *life-span* of any data point in the window on arrival — *i.e.* the future windows it will belong to. This leads to the notion of *predicted views*. Given the objects in a current window, they predict the pattern structures that will persist in subsequent windows by considering the objects in the current window and abstract these predicted pattern structures into *"predicted views"* of each future window. This technique allows the efficient maintenance of neighborship counts by pre-handling the eventual expiring effect of new data points.

In the paper, the authors first propose the Abstract-C and Abstract-M algorithms. Ultimately, they combine all the desirable properties of each algorithm into Extra-N.

3.2.2.1 Abstract-C

Abstract-C was the first proposed algorithm. The main idea is to maintain for each data point p_i the number of neighbors it has rather than the actual list of neighbors. The main advantage of

this approach is the reduced memory footprint since only a count is stored — a single integer — instead of a list of pointers — to all neighbors.

When a data point p_i expires, the count of each of p_i's neighbors must be decremented by one — since p_i no longer exists. However, we are unable to reach p_i's neighbors since in Abstract-C only a count of neighbors is stored — rather than pointers to each of the neighbors.

The solution to this problem is exploiting the previously presented *predictability* property. The key idea is to predict the expiration of any data point p_i and pre-handle the impact it has on p_i's neighbours. The authors introduce the concept of *"lifetime neighbour counts"* or *lt_cnt*. The *lt_cnt* of a data point p_i corresponds to a sequence of *predicted neighbour counts — i.e.* the number of *predicted neighbours* p_i has in any future window where he exists. More precisely, each entry on *pi.lt_cnt* records the number of p_i's current neighbours that are known to survive in the corresponding future window. Furthermore, the length of $p_i.lt\_cnt$ is kept equal to $p_i.lifespan$, decreasing by one after each window slide, removing the left most entry. The authors provide the following example: consider a window W_i and a data point p_i with three neighbours, p_1, p_2 and p_3. Applying the *predictability* property makes it possible to compute the lifespan of all points. Assume p_1 expires after W_i, p_2 and p_3 expire after W_{i+1} and p_i expires after W_{i+2}. This implies that at W_i, $p_i.lt\_cnt = \{W_i\text{: } 3, W_{i+1}\text{: } 2, W_{i+2}\text{: } 0\}$.

According to the definition given in 3.7, the lifetime neighbor counts (*lt_cnt*) carry enough information to compute distance-based outliers: for each data point p_i check if the first (current) element of $p_i.lt\_cnt$ is less than $\theta^{fra} \times N$ to decide whether a data point is an outlier or not, with N as the number of data points. Similarly, the core objects for the density-based clusters can be computed by comparing the first element of $p_i.lt\_cnt$ with θ^{count}. However, the *lt_cnt* array does not carry enough information to generate the density-based clusters. Despite being able to find all core points in the window, we can not group them into clusters. In order to compute that, Abstract-C analyzes each core point and the surrounding θ^{range} area in the window to reconstruct the clusters.

In conclusion, Abstract-C achieves linear memory consumption in the number of data points in the window. However, since it reconstructs the clusters for each core object in the window, it has *O(CP)* time complexity, with *CP* as the total number of core points. Hence, its performance largely depends on *CP*, which can vary from 0 – no core points — to the total window size — all core points.

3.2.2.2 Abstract-M

Abstract-M is proposed as an Abstract-C enhancement by introducing the notion of *cluster memberships*, achieved by attributing a *clusterID* to each data point.

The main challenge of this approach is discounting the effect of expired data points. The removal of any data point on any cluster may result in the total break of the cluster into multiple smaller ones, which may persist or degrade to noise. Furthermore, when a cluster is split, the newly formed clusters must be attributed with new *clusterID*s — *i.e.* every single data point in those clusters must be relabeled with the new *clusterID*.

Once again, the solution to handle expiring data points relies on the predictability property. Knowing the data points that exist in each window means we can also know the set of future clusters and points that belong. This is included in the *predicted views* concept previously mentioned. In brief, the *predicted clusters* are created using such predicted views for each of the future windows.

Addition of new data points may cause three changes to the clusters: *birth* of a new cluster, *expansion* of an existing one and *merge* of multiple existing clusters. When adding a new data point, mark it with a new *clusterID* in case of a *birth* or in an *expansion* case mark it with the *clusterID* of the cluster it belongs to. Handling the *merge* of multiple clusters is done by using a tree structure where two or more sibling *clusterID*s are in fact part of a bigger cluster with the *clusterID* stored in their parent. For example, if cluster *ID1* and cluster *ID2* are merged into a new cluster *ID3* by the arrival of a new data point, they are inserted as children of the node containing *ID3*.

New data points may join existing neighborhoods of existing data points and may promote them to core points by making the size of their neighborhood greater or equal to θ^{count}. Once such promotion happens, the promoted core point needs to communicate to its neighbors its new role, since noise in its neighborhood becomes edge points and is given a *clusterID*. Since there are no connections from the promoted data point to its neighbors, Abstract-M analyzes each of the promoted core points and the θ^{range} area around them to find the points requiring state update.

In conclusion, Abstract-M remains linear in memory regarding the number of window data points. However, since the number of promoted core points tends to be smaller and is always less or equal to the number of core points, this is an improvement on Abstract-C.

3.2.2.3 Extra-N

Abstract-M still performs range queries (area with θ^{range} diameter) for each of the promoted core points. To improve upon this, Extra-N combines the neighborship maintenance mechanisms proposed in Abstract-C and Abstract-M.

The authors state that different information is required from different classes of data points (*core*, *edge* or *noise*). For example, maintaining a list of points to neighbors is required for a non-core point while neighbor counts are sufficient for core points. More precisely, Extra-N marks each data point p_i with a *clusterID* in all windows where p_i is predicted to be a core point while keeping the exact list of neighbors (pointers) of p_i for all windows where he is predicted to be noise or an edge point. This way, the Extra-N algorithm stores the amount of information needed to compute both distance-based outliers and density-based clusters according to their definitions, 3.7 and 3.8 respectively.

3.2.2.4 Time complexity analysis

Extra-N only performs range queries for incoming data points. This means that for each new data point p_i, Extra-N will search the area of diameter θ^{range} around it and update on it neighbors. In

conclusion, the time complexity of Extra-N is *O(n)* where *n* is the number of new data points.

3.2.2.5 Space complexity analysis

In summary, Extra-N keeps the memory consumption linear to the number of data points in the window without performing unnecessary range queries.

3.2.2.6 Applicability to our Hypothesis

While a linear complexity fits most of the scenarios, it does not fit ours since we aim to build a low memory footprint solution, ideally with space complexity of *O(1)*. Hence, this algorithm will be of no use to us.

3.3 Anomaly Detection

Anomaly detection refers to the process of finding anomalies in data. An anomaly relates to a point in time where the behavior of the system does not match the expected one. Hence, under this definition, an anomaly does not necessarily imply a problem.

With the study of anomaly detection algorithms, we intend to search for algorithms that identify data pattern shifts as anomalies. If any of the approaches searched for is not computationally intensive it will be considered valid for the final solution.

3.3.1 Real-Time Stream Anomaly Detection with Hierarchical Temporal Memory

In [2], the authors present an anomaly detection technique based on an on-line algorithm called Hierarchical Temporal Memory (HTM). HTM has a few properties that are desirable for data stream anomaly detection scenarios. First, it is an unsupervised algorithm that requires no labels. This property comes in handy when monitoring a data stream in real-time, where labels for incoming data will likely be absent. Additionally, HTM adapts to noisy domains. In other words, random variations in the underlying distribution of incoming data that do not persist throughout time will be ignored. Furthermore, HTM detects both spatial and temporal anomalies — *i.e.* changes in magnitudes of data or unusual timing for particular patterns, respectively.

HTM is also an online learning algorithm, meaning that it adapts to changing statistics of the underlying data stream. In other words, if there is a sudden random increase in a specific data point attribute, the algorithm will produce an alert. However, if that initially thought to be a random spike becomes frequent then the algorithm adapts and assimilates this as "a new reality". Hence, it stops classifying it as an anomaly. Unfortunately, this property is not desirable in our system. Our stream monitoring system must report changes relatively to an initial static configuration and should not accept a new distribution as the new standard.

3.3.1.1 Applicability to our Hypothesis

Online learning algorithms that adapt to data pattern shifts and eventually consider them regular patterns are not viable methods to accomplish our goal of building a monitoring system that alerts data pattern shifts continuously until they match the initial configuration again. Additionally, insights produced by machine learning models are not human-understandable, rendering the user unable to understand why an anomaly report was made. In conclusion, we do not deem HTM applicable to our solution.

3.3.2 DeepAnT

DeepAnT [24] is a deep learning-based approach for the detection of anomalies in time series data, that uses unlabeled data to capture and learn the data distribution of the stream used to forecast the normal behavior of the time series data. It is unsupervised - *i.e.* it requires no labels - which is an advantage in a data streaming scenario where real-time decisions have to be made before labels are effectively-known. It is capable of detecting point anomalies in time series data with periodic and seasonal characteristics.

DeepAnT employs a Convolutional Neural Network (CNN) as its *time series predictor* module. Before monitoring a data stream, the CNN must be trained. The model can be trained with a small data set while achieving good generalization capabilities due to the effective parameter sharing of the CNN. Additionally, the data set used to train the model thus not require labels and may up to 5% of the data set may correspond to anomalous data.

3.3.2.1 Applicability to our Hypothesis

Similar to the HTM algorithm presented, DeepAnT adapts to the underlying distribution of the data stream. In other words, if the data patterns shift from the initial configuration but stabilize, DeepAnT will accept it as it is. As mentioned, our goal is to build a monitoring system that alerts data pattern shifts continuously, until they match the initial configuration again. Furthermore, DeepAnT uses deep learning which is actually a subset of machine learning and just as inexplicable. In summary, we do not deem DeepAnT usable for our use case.

3.4 Probabilistic Data Structures

Probabilistic data structures use hash functions to compactly represent a set of items while approximately answering queries with fixed error bounds. This way, these structures are capable of processing huge volumes of data. They require a single pass through the data, which is appropriate for a streaming scenario. Probabilistic data structures have constant space and time complexity [26] making them candidate methods to build our real-time and low memory footprint system.

3.4.1 Membership Queries and the Bloom Filter

A Bloom filter [9] is a probabilistic data structure that allows membership queries on a set of elements. Bloom filters answer the query: "Is element *el* in the set of seen elements so far?". Being a probabilistic data structure, Bloom filters do not always give a certain answer. *False positive* matches are possible — *i.e.* saying *el* is in the set when it is not — but *false negatives* are not — *i.e.* saying *el* is not in the set but it actually is. In other words, a bloom filter will accurately identify all items that do not belong in the set but will misclassify some items as being present in the set. Hence, a query returns either *possibly in set* or *definitely not in set*.

A Bloom filter is an array of bits of size m. A Bloom Filter makes use of k hash functions. The choice of the m and k constants will determine the false positive rate.

Initially, all bits are set to 0. Adding an element is done by determining which positions should be set to 1. To that end, the new element is fed into each hash function that maps the element to an array position. The resultant k outputs are used as the k positions of the array to be set to 1.

Performing a membership query — *i.e.* testing if an element *el* is in the set — is done by feeding *el* into each one of the k hash functions in order to get an array of positions. If *el* was already in the set then all k positions should be 1. If any position contains a 0 then the element is *definitely not in set*, as shown in Figure 3.9 from [7]. If all positions are 1 then the element is *possibly in the set*.

The reason a Bloom Filter does not provide 100% certainty that *el* is in the set in case of all positions being 1 is that hash functions may give the same position for two different elements. For example, hash functions k_1, k_2 and k_3 may map an element *el1* to positions $[1,2,3]$ and an element *el2* to positions $[4,5,6]$. This way, bits in positions $[1,2,3,4,5,6]$ are set to 1. Given a new element *el3* not yet seen, hash functions k_1, k_2 and k_3 may map it to $[1,3,5]$ and all of these positions are already 1 because of the previous elements. Despite all bits being set to 1, *el3* was not in the set. This is considered a *false positive*.

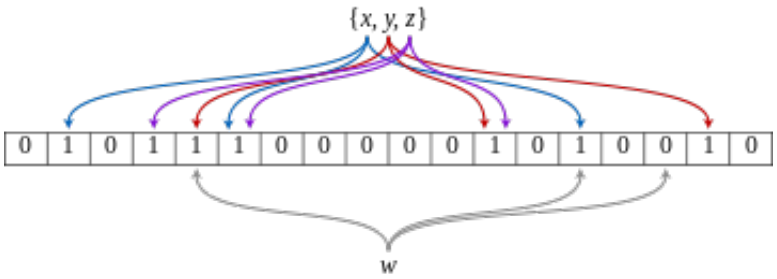

Figure 3.9: Bloom filter membership test of element 'W'. Finding one '0' indicates that it is not in the set.

An improvement to this structure was defined in [20] where the authors attempt to reduce the overhead of bloom filters operations such as adding a new element or answering a query by focusing on the hash functions. The proposed method is based on hashing literature. It has been shown that multiple hash functions can be created from the combination of only two hash

functions. This means that a new hash function g can be produced based on two other independent and uniform existing hash functions, *h1* and *h2*. The new hash function will map elements to a universe ranging from *0* to *p-1* and will be defined as $g(x) = h1(x) + ih2(x) mod p$. Since the only nontrivial computations performed by a Bloom filter are the evaluations of pseudo-random hash functions, any reduction in the required number of pseudo-random hash functions yields a reduction in the time required to add an element or query a Bloom filter.

Bloom filters are not the only approximate aggregators that use hash functions as a basis of their algorithm. All of the studied probabilistic data structures do so. Therefore, the technique proposed is beneficial in HyperLogLogs, Count-Min Sketches and Bloom filters.

3.4.1.1 Time complexity analysis

Considering a Bloom filter with m bits and k hash functions, insertion and search will both take *O(k)* time because all there is to do is run the input through all of the k hash functions and set the bits in the given positions to 1. Note the time complexity does not at all depend on the number of elements in it and that k will be a rather small constant (magnitude of 10^1). Hence, the Bloom filter is said to have constant time complexity (*O(1)*).

3.4.1.2 Space complexity analysis

Considering a Bloom filter with m bits, the space required is simply the array of m bits, thus *O(m)* space complexity. Similarly to the time complexity analysis, we point out that m will be constant and that each of the array elements occupies 1 bit. Thus the Bloom filter has a constant space complexity (*O(1)*).

3.4.1.3 Applicability to our Hypothesis

Bloom filters exhibit constant space and time complexity making it great methods to test our hypothesis. Their false positive rate is tolerable in most scenarios and can be controlled by tweaking the values of m and k.

3.4.2 Item Frequency and the Count-Min Sketch

A Count–Min Sketch (CMS) is a probabilistic data structure that gives an estimate of the frequency of each element in the data set. The estimate is a minimum count. The real frequency of an element will never be lower than the given estimate, hence the name *count min*. The CMS is first introduced in [10].

A CMS is represented by a two-dimensional array (or matrix) with width w and depth d. A CMS will make use of d hash functions — *i.e.* each associated with one row of the matrix. Each hash function will map elements to a position between 1 and w — *i.e.* the column of the two-dimensional array.

Initially, all of the matrix positions are 0. When adding an element, for each row i of the d rows in the matrix, hash the element using that row's hash function to obtain j. Lastly, for all obtained pairs of i and j, increment matrix cells (i, j) value by 1, as seen in Figure 3.10.

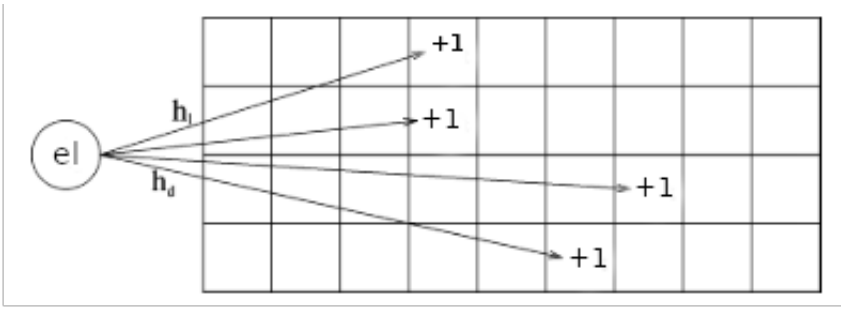

Figure 3.10: CMS mapping element *el* using *d* hash functions to determine the column position for each row and increment it

Retrieving the minimum frequency of an element *el* can be done by taking the minimum value of all row counts for *el*. Mathematically, the frequency estimate is given by

$$min\{matrix[i][h_i(el)]\}$$

for each row i of the total d rows.

3.4.2.1 Time complexity analysis

When adding an element or querying the frequency of one, the Count-Min Sketch loops through each row and applies a constant in time hash function, proceeding to a position update or retrieval, respectively. Hence, the time complexity is of $O(d)$. Given d is a constant, the CMS has constant time complexity — $O(1)$.

3.4.2.2 Space complexity analysis

The Count-Min Sketch relies on a two-dimensional matrix as its only data structure. The memory used will correspond to the wd counts, hence $O(wd)$ space complexity. Since both w and d are constants, the CMS has constant space complexity — $O(1)$.

3.4.2.3 Applicability to our Hypothesis

Count-Min Sketches are constant in both time and space. Similarly to Bloom filters, this makes them ideal candidates for our solution.

3.4.3 Cardinality Estimation and the HyperLogLog

In the paper [13], a class of probabilistic counting algorithms is introduced to estimate the number of distinct elements based on *"bit pattern observables"* in the binary representation of the hashed values — *i.e.* patterns related to the number and positions of 0's and 1's in a binary string.

3.4.3.1 LogLog

Later on, the same authors propose the LogLog algorithm [11]. In LogLog, the *bit pattern observable* recorded from each item's hash value is the position of the leftmost 1-bit. The authors claim it is related to the total number of distinct elements in the data set.

3.4.3.2 HyperLogLog

A couple of years later, the same authors propose HyperLogLog (HLL) in [12]. HLL solves the problem of counting distinct elements in a data set, also known as the cardinality of the data set [12]. HLL gives an approximation of the cardinality of the data set.

In HyperLogLog, an item is hashed into a binary string. Based on the first n bits, the string is passed on to one of the 2^n *buckets*. After discarding the first n bits, the number of trailing 0's plus 1 is counted. In other words, the position of the leftmost 1-bit is recorded, like in LogLog. This is what is stored in each bucket: the maximum value of the position of the leftmost 1-bit of all seen items so far.

Finally, the estimate for the number of distinct elements will be 2^p, where p is the harmonic mean of all bucket values holding the maximum leftmost 1-bit position value. This computation of a harmonic mean over m buckets is called stochastic averaging and is the main difference between HyperLogLog and LogLog. Given the nature of streaming systems, it is also important to notice that this algorithm is suitable for a distributed system since each of the buckets can be updated independently.

3.4.3.3 Time complexity analysis

Adding an item is tantamount to hashing it, determining the corresponding bucket based on the first n bits, count the leftmost 1-bit position and update the bucket value if need be. Estimating the cardinality of the set amounts to finding the maximum value p between all buckets and computing 2^p. Hence, both inserting a new element and estimating the cardinality take constant time — *O(1)*.

3.4.3.4 Space complexity analysis

Quoting Flajolet et al. about the memory consumption and error in the approximations obtained in their experiments in [12]: "cardinalities till values over N = 10^9 can be estimated with a typical accuracy of 2% using 1.5kB (kilobyte) of storage."

HyperLogLog uses m buckets that store a single integer value. This gives it complexity of *O(m)* where m is a small constant. Hence, HyperLogLog has constant space complexity — *O(1)*.

3.4.3.5 Applicability to our Hypothesis

Similar to the previously studied probabilistic data structures, HyperLogLog (HLL) has constant space and time complexity while giving an approximation of a set's cardinality. Hence, HLLs are considered valuable methods for our work.

Chapter 4

Problem and proposed solution

4.1 Scope

The focus of this Thesis is the development of a lightweight real-time data monitoring system working in the presence of high volumes of high velocity, high variety, highly skewed and seasonal data. The resultant system should detect and report data pattern shifts in an unsupervised fashion.

Essentially, the system will summarize incoming data with sliding window aggregations and probabilistic data structures in a sliding window fashion and compare current feature values with an initial given static configuration. Any big enough deviations between these two states should be reported as a data pattern shift.

Furthermore, the resulting system shall strive to achieve high precision and recall ratios as well as low latencies with a low memory footprint.

4.2 Research Questions

With this Thesis, we intend to build a system that monitors real-time data streams in a lightweight fashion. As such, in the end, to determine if the system is successful in doing so or not, the following research questions must be answered:

RQ1: Can the system accurately detect data pattern shifts — *i.e.* does it have **high precision ratio**?

RQ2: Can the system detect most of the data pattern shifts — *i.e.* does it have **high recall ratio**?

RQ3: Can the system detect data pattern shifts in real-time — *i.e.* does it work under extremely **low latencies**?

RQ4: Can the system detect data pattern shifts in a lightweight fashion — *i.e.* does it have a **low memory footprint**?

4.3 Experimental Methodology

In order to validate or reject our proposed hypothesis, a set of tests will be carried out to determine the system's usefulness. The data sets used will belong to the financial fraud space. The output of the system will be compared to the results of an existing batch analysis tool, for the same data sets, both provided by Feedzai.

To test the stream monitoring system we will need a streaming engine. As such, either a custom-built streaming engine or an existing one — *e.g.* like Apache Flink [25] — will be used.

In brief, it is desired that our in real-time system monitoring report is as close to the batch analysis in terms of accuracy as possible.

4.4 Planning

To accomplish the proposed experiments, the following steps are required:

T1: **Research more state of the art aggregations** - search for more memory efficient sliding window aggregations to use in the final system.

T2: **Data set and batch results analysis** - explore chosen data sets and the resulting batch analysis made on them.

T3: **Stream Processing Engine (SPE) setup** - setup the stream processing engine chosen — *e.g.* Apache Flink [25] — or build a custom very basic one.

T4: **Implementation of chosen algorithms** - implement the selected algorithms in an efficient programming language — *e.g.* Java or C++ — in a sliding window fashion and integrate it with the SPE.

T5: **Test the implementation with the chosen data sets** - compare the batch analysis with our streaming analysis and measure precision and recall ratios as well as latency and memory usage.

T6: **Thesis writing** - the writing of the Thesis document will be an on-going process but ultimately finished last.

In Image 4.1 we present the expected start and end dates for the main tasks presented before.

Furthermore, we built a Gantt Chart 4.2 that illustrates the expected time frame of completion for each task, in weeks.

Task	Start	Finish
T1	17 Feb 20	01 Mar 20
T2	02 Mar 20	08 Mar 20
T3	09 Mar 20	16 Mar 20
T4	23 Mar 20	20 Jul 20
T5	23 Mar 20	20 Jul 20
T6	13 Apr 20	03 Aug 20

Figure 4.1: Tasks and deadlines

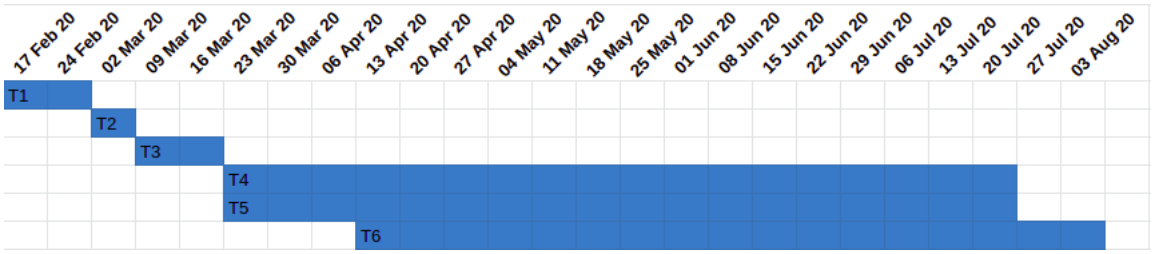

Figure 4.2: Gantt Chart of Thesis research plan

Chapter 5

Conclusion

We introduced background concepts on large scale data processing, windowing techniques applied to data streams and sliding window aggregations. We studied algorithms for change detection in data streams, pattern mining, anomaly detection and sliding window aggregation algorithms, specifically a subset of these named probabilistic data structures.

The monitoring of data streams in real-time is applicable across many domains and applications. To the best of our knowledge, we claim that there has not been a real attempt at doing so in a lightweight way.

Preliminary work shows that there exist constant space and time complexity algorithms to aggregate data, thus reducing memory usage. We intend to research more about lightweight sliding window aggregations and sliding implementations of probabilistic data structures. We also intend to search for new methods of summarizing data that do not necessarily explicitly work in a sliding window fashion. After identifying a set of algorithms that meet our requirements, the focus will be on implementing the best theoretical solution and benchmark our streaming system using data sets from the financial fraud space versus previously performed batch analysis.

The main contribution of this Thesis will be the developed system code as well as a set of tests that measure the system's usefulness based on several parameters, such as recall, precision, memory used and latency.

References

[1] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, page 81–92. VLDB Endowment, 2003.

[2] Subutai Ahmad and Scott Purdy. Real-Time Anomaly Detection for Streaming Analytics. *CoRR*, 2016.

[3] Apache flink. https://flink.apache.org/. Accessed: 2020-01-23.

[4] Apache hadoop. http://hadoop.apache.org/. Accessed: 2020-01-23.

[5] Apache spark. https://spark.apache.org/. Accessed: 2020-01-23.

[6] Batch is a special case of streaming. https://www.ververica.com/blog/batch-is-a-special-case-of-streaming. Accessed: 2020-01-30.

[7] Bloom filter wikipedia. https://en.wikipedia.org/wiki/Bloom_filter. Accessed: 2020-02-10.

[8] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J. Miller, and Nesime Tatbul. SECRET: A model for analysis of the execution semantics of stream processing systems. *Proceedings of the VLDB Endowment*, 3(1):232–243, 2010.

[9] BURTON H. BLOOM. Space/Time Trade-offs in Hash Coding with Allowable Errors. *ACM SIGDA Newsletter*, 8(1):6–11, 1978.

[10] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The Count-Min Sketch and its applications. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2976:29–38, 2004.

[11] Marianne Durand and Philippe Flajolet. LogLog Counting of Large Cardinalities. *inproceedings*, pages 1–14, 2007.

[12] Philippe Flajolet. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *2007 Conference on Analysis ofAlgorithms, AofA 07*, pages 127–146, 2001.

[13] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[14] Joao Gama. *Knowledge Discovery from Data Streams*. Chapman & Hall/CRC, 1st edition, 2010.

[15] Bugra Gedik. Generic windowing support for extensible stream processing systems. *Software - Practice and Experience*, 39(7):701–736, 2009.

[16] Chris Giannella, Jiawei Han, Xifeng Yan, and Philip S Yu. Mining Frequent Patterns in Data Streams at Multiple Time Granularities. *Next generation data mining*, pages 191–212, 2003.

[17] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(2):1–12, 2000.

[18] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, page 97–106, New York, NY, USA, 2001. Association for Computing Machinery.

[19] D KIFER, S BENDAVID, and J GEHRKE. Detecting Change in Data Streams. *Proceedings 2004 VLDB Conference*, pages 180–191, 2004.

[20] Adam Kirsch and Michael Mitzenmacher†. Less Hashing, Same Performance: Building a Better Bloom Filter. *Random Structures and Algorithms*, pages 524–548, 2006.

[21] Taiwo Kolajo, Olawande Daramola, and Ayodele Adebiyi. Big data stream analysis: a systematic literature review. *Journal of Big Data*, 6(1), 2019.

[22] Niels Martin, Marijke Swennen, Benoît Depaire, Mieke Jans, An Caris, and Koen Vanhoof. Batch processing: Definition and event log identification. *CEUR Workshop Proceedings*, 1527:137–140, 2015.

[23] Amaryllis Mavragani, Gabriela Ochoa, and Konstantinos P Tsagarakis. Assessing the methods, tools, and statistical approaches in google trends research: Systematic review. *J Med Internet Res*, 20(11):e270, Nov 2018.

[24] Mohsin Munir, Shoaib Ahmed Siddiqui, Andreas Dengel, and Sheraz Ahmed. DeepAnT: A Deep Learning Approach for Unsupervised Anomaly Detection in Time Series. *IEEE Access*, 7:1991–2005, 2019.

[25] Tilmann Rabl, Jonas Traub, Asterios Katsifodimos, and Volker Markl. Apache Flink in current research. *it - Information Technology*, 58(4):157–165, 2016.

[26] Amritpal Singh, Sahil Garg, Ravneet Kaur, Shalini Batra, Neeraj Kumar, and Albert Y. Zomaya. Probabilistic data structures for big data analytics: A comprehensive review. *Knowledge-Based Systems*, page 104987, 2019.

[27] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. Sliding-Window Aggregation Algorithms. *Encyclopedia of Big Data Technologies*, pages 1516–1521, 2019.

[28] Twitter throughput. https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.html. Accessed: 2020-02-1.

[29] Di Yang, Elke A. Rundensteiner, and Matthew O. Ward. Neighbor-based pattern detection for windows over streaming data. *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, EDBT'09*, pages 529–540, 2009.