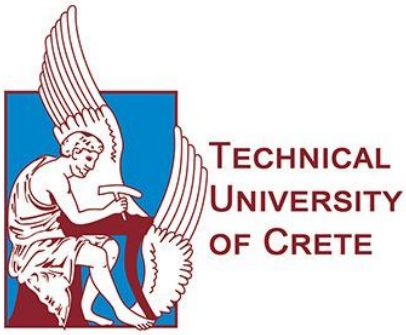
Technical University of Crete
School of Electrical and Computer Engineering



An Engine for Efficient Data Stream Summarization using Kafka and Kafka Streams Microservices

MASTER THESIS BY

Georgios Panagiotis Kalfakis

Thesis Committee:

☛ **Supervisor:**

Prof. Giatrakos Nikos

☛ **Advisors:**

Prof. Deligiannakis Antonios

Prof. Garofalakis Minos

Date: May 27, 2024

AN ENGINE FOR EFFICIENT DATA STREAM SUMMARIZATION USING KAFKA AND KAFKA STREAMS MICROSERVICES

ACKNOWLEDGEMENT

I wish to thank various people for their contribution to this project and without their support the completion of this thesis would not be possible. First of all, I would like to express my deep gratitude to **Asst. Professor Nikos Giatrakos**, my research supervisors, for their patient guidance, enthusiastic encouragement, and useful critiques of this research work. I would also like to thank **Professor Antonios Deligiannakis** and **Professor Minos Garofalakis**, for their advice and assistance in keeping my progress on schedule. My grateful thanks are also extended to the cluster administrator, **Dr. Arapi Xenia**, for her help in setting up the infrastructure used in the experimental evaluation of this thesis. Finally, I wish to thank my parents for their support and encouragement throughout my studies.

ABSTRACT

In this work, we introduce a novel stream summary maintenance paradigm in the form of distributed microservices, namely Synopses as a MicroService, and we implement this paradigm on top of Apache Kafka and Kafka Streams Microservices. SaaMS is designed for real-time stream summarization and analysis over rapid data streams. SaaMS also contains a built-in library with Synopses that is used for producing stream summaries but remains extensible and customizable to new Synopses techniques. In that, (a) it contributes an innovative architecture to gain scalability dynamically based on the necessary computation requirements, (b) maintains a large volume of Synopses, concurrently with high throughput and fault-tolerance, (c) provides an extensible Synopsis library for real-time analysis (d) experimental evaluation provided using real financial data. SaaMS manages large-scale stream processing and analysis because it enables (i) horizontal scalability, i.e., taking advantage of complicated mechanisms that Kafka has for distributing the workload, achieving maximum throughput and minimum latency (ii) vertical scalability, i.e., the ability to scale the computation with the number of processed streams (iii) federated scalability, i.e., data can be processed across multiple distributed environments even in case they are geographically dispersed.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

The ever-increasing demands for real-time analytics and the high-velocity flow of data are affecting more and more domains. It has changed the way data is processed and managed. Since we live in the era of **"Big Data"**, this is forcing more and more industries to change the way they operate. In the financial domain[33], in previous decades it was necessary to store and analyze vast quantities of data to identify commonalities which could be disastrous in many cases. This has changed with the use of streaming data, as a result, we can combine historical data with the real-time trends of financial markets to predict patterns and plot their next moves, providing a highly accurate tool for investors and advisors. This type of analysis can also be useful for regulators to detect potential signs of fraudulent behavior, a common problem in stock markets. The use of real-time processing in stocks will become more widespread with the further development of artificial intelligence used to predict stock performance. Additionally, the use of data streams is increasingly being adopted in the agricultural industry [22] because it requires managing data processed by sensors in real-time in an effective automated way. Improving how spraying, seeding, and harvesting are done, as well as weather prediction, this technology will help to address potential problems in cultivation. The rapid changes brought about by climate change will only increase this necessity in the future. Real-time processing is also implemented in healthcare [31] for improving the health system overall. More specifically, it offers the opportunity to build integrated systems that contain Electronic health records, Electronic prescription services, and Health-related smartphone apps. These systems improve the services provided to citizens both at the level of diagnosis and at the level of convenience. Furthermore, the cost of many functionalities is decreasing due to automation. All these examples present a little part of the uses of stream processing and it is certain that in the future it will be an even more necessary tool in industry.

The systems managing large-scale stream processing and analysis functionalities must provide three types of scalability:

- Horizontal scalability is the ability of a stream application to scale effectively to demands by adding more nodes. This way helps to handle extreme data volumes by parallelizing the workload to increase the computation capabilities. The distribution operation improves also fault tolerance and availability, as well as performance.

- Vertical scalability in general is the capability of expanding the processing power of a system to handle the increasing workload. In this thesis, however, we are going to refer to vertical scalability as the ability to scale the computation with the number of processed streams.

- Federated scalability is the capability of a system to scale data across different environments that are geographically dispersed. This means that managing data from multiple locations leverages a wider array of data sources.

There are several Big Data platforms [34] for real-time processing which can provide only the necessary horizontal scalability out of the 3 scalability types mentioned above e.g. Apache Kafka Streams, Apache Flink, Apache Spark, etc. The best option to build a project differs based on the use case. In extreme scale scenarios, also it is required, in addition to processing the data with a suitable stream processing framework, to summarize the data using different techniques such as samples[6], sketches [2] or histograms[28]. The synopsis techniques are designed in such a way that they can provide the aforementioned 3 types of scalability by performing the computation on carefully crafted data summaries. In that, synopses scale the computation by combining parallel processing and stream summarization controllably sacrificing accuracy in analytics, with predefined accuracy guarantees. As a result, the combination of real-time processing and analysis provides in a short time answers for continuous data streams.

In this work, we introduce a new synopses maintenance paradigm, namely Synopses as a MicroService (SaaMS). SaaMS is a stream synopses computation and maintenance paradigm realized in a Kafka Streams implementation to adopt the necessary scalability as we already mentioned and execute scalable real-time stream processing and analysis with specific summarization techniques. SaaMS is built using the Kafka [3] and Kafka Streams [4] framework to handle the real-time pro-

cessing requirements. SaaMS provides a built-in library of a wide variety of stream summarization techniques, simultaneously retaining the ability to customize it by plugging in its library of new synopses techniques.

1.1. Motivation

The ever-changing technological landscape, how we manage Big Data but also the need for real-time analysis are the reasons for this thesis. More specifically, in the era of Big Data, as we saw in the examples (healthcare system, financial market, and agriculture fields), real-time analysis for better decisions and results is required. Delving further, it was clear that there is a lack of Big Data frameworks offering native tools to implement real-time stream summarization. On the other hand, in the literature, there exist a lot of synopsis libraries [35] [26] which contain algorithms working on data streams but don't have the necessary scalability, because they are detached from Big streaming Data processing frameworks. Based on this observation, we introduced and implemented SaaMS to provide scalability (horizontal, vertical, federated) and to maintain on-the-fly real-time analytics for a wide variety of commonly used application queries. In the context of financial markets which is the use case we use in our experimental evaluation, SaaMS is a dynamic application adapted to receive data from financial markets and perform real-time processing and analysis to locate anomalies, insights, etc. enabling correct decision-making.

1.2. Thesis Contribution

The contributions of this thesis are:

1. We introduce SaaMS, a robust **innovative architecture** to gain scalability in all dimensions which can adjust dynamically based on the necessary computation needs. In particular, it can provide high throughput (number of tuples being processed per time-unit) and scale up or down with zero downtime for the running queries. Also, SaaMS can provide an automatic handling of the addition of new instances for geographically dispersed computing nodes.

2. **SaaMS maintains a large volume of Synopses, concurrently** without reductions in performance, with high throughput and fault-tolerance. Each Synopsis is an independent

microservice running in multiple parallel instances and can handle data from different or the same source.

3. SaaMS provides a **Synopsis library** which contains different summarization techniques used for real-time analysis. This library can be easily extended due to the way it is designed based on inheritance and polymorphism.

4. **SaaMS experimental evaluation** using real data from the financial domain with high volume from different sources providing summations.

1.3. Outline

➢ Chapter 2 provides useful theoretical background on summarization techniques, Kafka and Kafka Streams

➢ Chapter 3 describes the functional and non-functional requirements of SaaMS application

➢ Chapter 4 analyzes the implementation details of SaaMS architecture, the Synopsis library, and the user interface

➢ Chapter 5 experimentally analyzes the performance of SaaMS

➢ Chapter 6 recaps about SaaMS and discusses future work

CHAPTER 2

BACKGROUND

2.1. Introduction

The world of data has changed vastly over the last decade. The main reasons are the exponential growth in the volume and velocity of the data, but also the increasing need to analyze streaming data in an online real-time fashion to serve applications that rely on timely and continuous analytics answers to fulfill their business goals. This situation makes databases and offline analytics approaches unable to fit out an efficient solution due to the static characteristic of collecting an amount of data and processing it in batches. The necessity for more drastic solutions has created stream processing[32], which is a quite different way of processing and analyzing data compared to traditional batch processing. Stream processing collects in real-time a continuous flow of data with the main characteristics of volume, velocity, and the requirement for scalability. Scalability is required to avoid high computational latency and low throughput in both processing and continuous querying procedures. For a better understanding of the differences between a Data Stream Management System (DSMS) and a Database Management System (DBMS), the following table is presented:

DSMS (Data Stream Management Systems)	**DBMS (Database Management Systems)**
Handles volatile data streams	Deals with persistent data (relations)
Sequential access	Random access
Continuous queries	One-time queries
Limited main memory	Unlimited secondary storage
Considers the order of input	Focuses on the current state
Potentially high update rate	Relatively low update rate
Real-time requirements	Little or no time constraints
Assumes potentially outdated/inaccurate data	Assumes exact data

Table 2.1: Comparison between DSMS and DBMS (by Wikipedia [39])

The modern-day needs have made it necessary to create frameworks that are designed to manage and process data in real-time with high scalability and for distributed datasets on a large scale. These frameworks are suitable for data analysis and decision-making applications, where repercussions of delays are significant. Another advantage of these platforms is that they can leverage clusters with many computers to distribute the workload while also addressing failures making processing fault-tolerant. There are several options for Big Data platforms to build an application on like Apache Spark, Flink, and Storm which are good choices if your primary focus is on data stream processing and analysis. On the other hand, if it is necessary to build a project with scalability, durability, and high throughput for large-scale data streams, that can not only run on powerful computer clusters but also on any Java-enabled device, Apache Kafka[3] is the best solution. Taking into account that Kafka can cooperate with Kafka Streams[4], which is a library that extends the messaging capabilities of Kafka, this reduces the need to use another framework like Spark, Flink, etc., for processing data across the data source to cloud continuum.

Having already briefly introduced the capabilities of Apache Kafka and Kafka Streams for real-time processing, it's also important to address how data streams can be analyzed and summarized. Data summarization[30] is a set of well-known techniques in real-time processing for an efficient analysis of continuous streams providing approximate answers to commonly used analytics queries trading off accuracy for processing speed and communication reduction. The advantage of processing and simplifying continuously flowing data streams with high volume into actionable information in a short time makes it the best approach for timely decision-making. Continuing with data summarization analysis, there are two types of summaries, numerical and visual. We will focus on numerical summaries, which use statistics to understand properties of data (like averages), sampling methods(such as ChainSampler[6]), and detect patterns that may exist. A deeper view of how Kafka and Kafka Streams work and more about data summarization will follow in the next sections of this chapter.

2.2. Kafka and Kafka Streams

2.2.1. Apache Kafka

In this section, the basic structural features[13],[16],[14] of Apache Kafka will be described:

Kafka Cluster

A Cluster in Apache Kafka is a group of computers within a computer cluster working together to handle large volumes of data while keeping low latency, durability, and scalability. An overview of the tools and capabilities of a Kafka cluster includes fault tolerance and high availability using data replication, the organization of a group of brokers, leaders, and followers in topics, load balancing and scalability for achieving maximum throughput, and finally Zookeeper Integration for managing and coordinating the Kafka cluster.

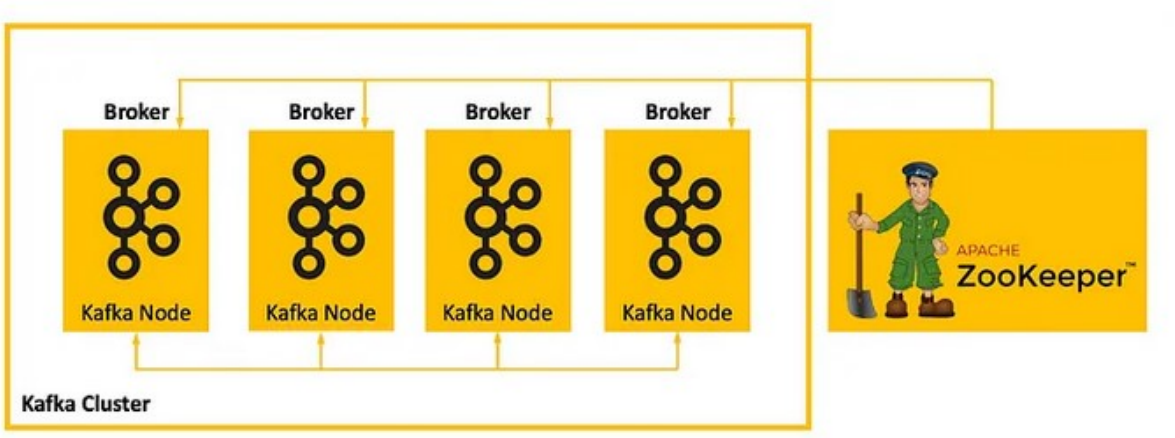


Figure 2.1: Kafka Clusters Architecture[36]

Kafka Logs Broker and Topics

Apache Kafka is a well-known distributed, scalable, elastic, and fault-tolerant event streaming platform. It is a type of file system and message queue storing an immutable amount of data in the form of an advanced message queue. The broker server is a storage component of Kafka responsible for hosting logs and organizing data efficiently. Logs are named and categorized by topics similar to directories in the broker's file system. Kafka topics are similar to a database but simpler, as newly

arrived messages are added to the end. Based on the real-world examples provided in Chapter 1 in the financial domain, Kafka topics can store various aspects of financial data such as stock prices, trading volumes, or transaction records in, for instance, a per stock or market fashion in each topic. In the agricultural industry, Kafka topics can be utilized to capture sensor data on weather conditions, crop growth, etc. In healthcare, Kafka topics can be used to organize and process data from electronic health records or electronic prescription services.

Kafka Partitioning and Replication

Partitioning is a functionality of Kafka topics that increases scalability and distributes the workload of storing, writing, and processing messages. A key field is the main factor in defining how data will be shared across partitions. Securing fault tolerance requires replicating each partition to more than one broker to ensure that in case of trouble, we don't have failures and lose records. To function we need to use a system called the Leader-Follower Model, each partition has a leader who manages all the requests for consuming and producing data and a follower trying to be in sync with the leader by replicating messages. In the case the leader fails for any reason, automatically one of the followers is elected as the new leader. The Leader-Follower Model enhances consistency and fault-tolerance across the partitions.

Kafka Consumer and Producer

Kafka can send and receive data using producers and consumers. A producer is a data source that initializes input by sending records to the log and automatically each record is assigned a unique 'offset', which is a value that determines its position in the log. On the other hand, consumers send a fetch request to read records using the offset as a pointer to identify where they previously stopped. A main characteristic of Kafka consumers is the organization into groups, allowing the distribution of partitions across all the groups and the consumers of a single group.

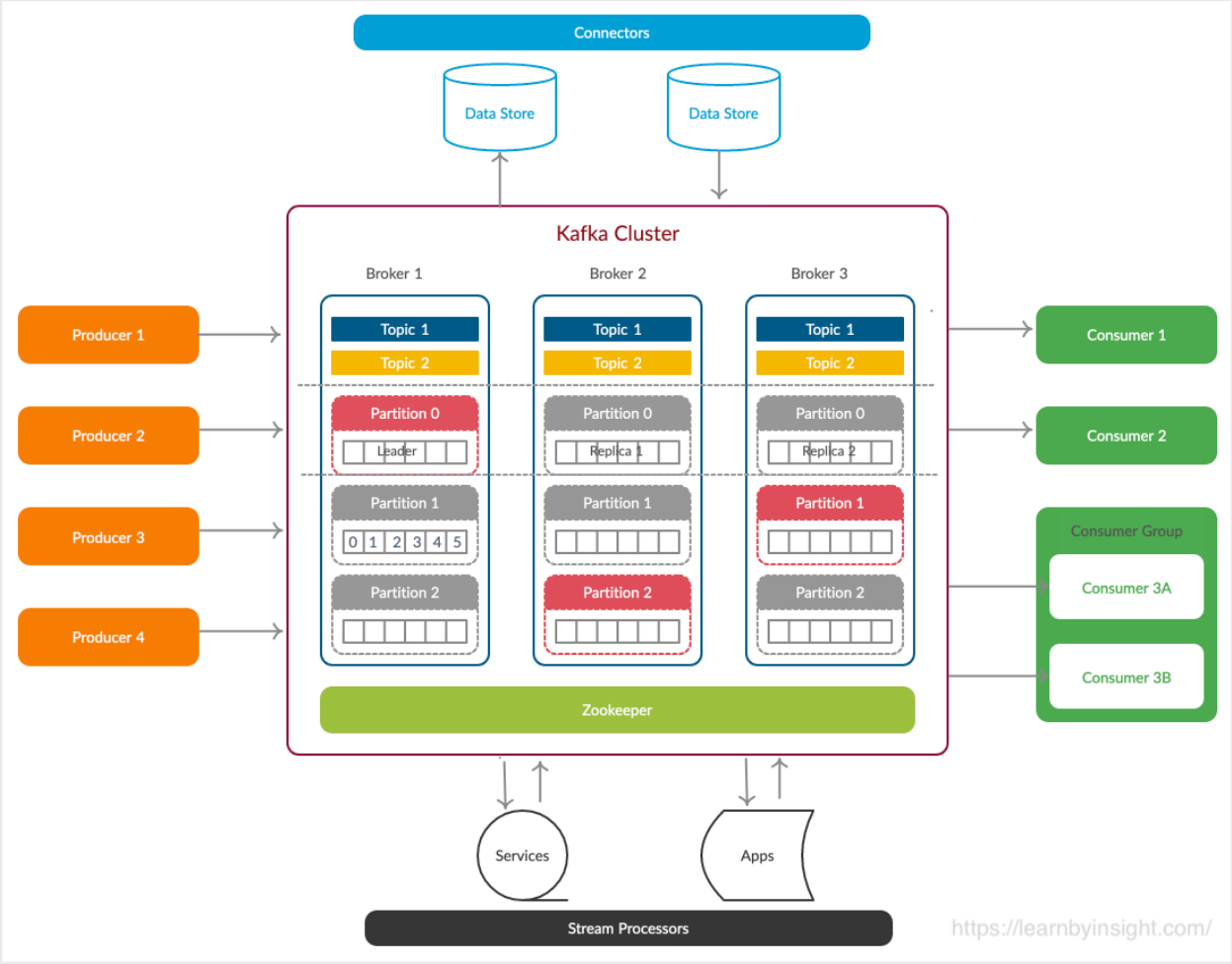


Figure 2.2: All the major components of a Kafka Clusters Architecture[16]

2.2.2. Kafka Streams

Kafka streams[9],[10] provides a far more concise approach to the functionalities than the vanilla Kafka without the need to use a low-level interface. In the following paragraphs, we will describe all the processing features that are required to understand SaaMS operations. To begin with, Kafka Streams are organized in key-value pairs of records. It is important to ensure the independence of each record even if they have the same key. This technique guarantees the accuracy and consistency of data processing. In the following, we present the basic components of a Kafka Streams framework:

Topology

Topologies are responsible for the flow of stream processing and are represented as directed acyclic graphs (DAGs). The basic components of topologies are sources (read data from Kafka topics), processors (perform transformations on this data), and lastly sinks (write the processed data to Kafka topics)

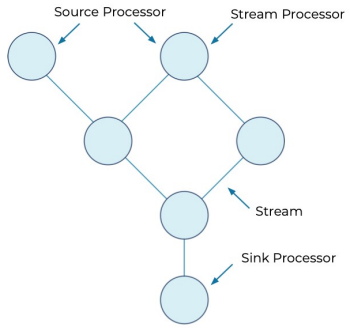


Figure 2.3: Topologies Architecture[9]

Core Concepts and Stream Transformation Operators in Kafka Streams

We have already analyzed how data is produced and consumed in Kafka, so it is necessary to focus on some basic operations for processing data.

- KStream and KTable are abstractions that represent different types of data flows. The first one is used to store continuous streams of records and the second one to store the last update value for a specific key. They are the most important elements for designing are Kafka Streams application due to the impact of how data is processed, stored, and translated.

- StateStore is an important component in maintaining and managing state information of records. It can be implemented by RocksDB (an embedded database for storing key-value pairs) or in memory. The use of a StateStore contributes to the scalability and fault-tolerance of Kafka Stream applications due to it can be distributed across multiple instances to handle the efficiency of the workload. For example, the StateStore in the financial market scenario can be used to provide information such as the current stock price or summary, trading volume, and patterns.

- Mapping operation: transforms an input object to an output object of another type. Map Values and Filtering are well-known functions.

- Serialization: The Kafka broker recognizes only bytes, so it is important to use Serialization in the Kafka Streams application. Serialization is the term used to describe data conversion from a specific type into an array of bytes and Deserialization is the opposite, refactoring an object from an array of bytes.

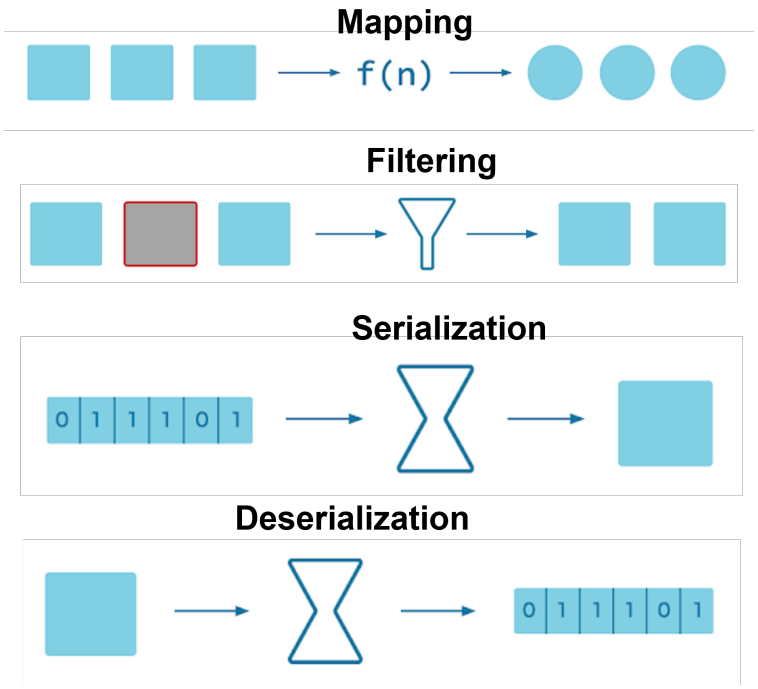


Figure 2.4: Basic Kafka Streams Transformations [9]

- Stream-Stream joins: Stream-Stream join merges two different streams into one based on a key in a specific time window and uses a change log topic for buffering. There are 3 types of joins, for Inner joins both sides must have the same key to produce the new merged record, outer joins always produce a record even if the key doesn't match in this case, we join one side with null. Finally, the left outer join has a similar logic with an outer join but for the records of the left side keeping them all and matching with the right side if the key is equal, else with null.

- Stream-Table joins, this type of join has a significant difference from Stream-Stream joins in that they are non-windowed (i.e., cannot be applied on windows of streaming records) and driven by the stream side. Inner and left joins work the same as Stream-Stream joins. However outer join doesn't exist in this type.

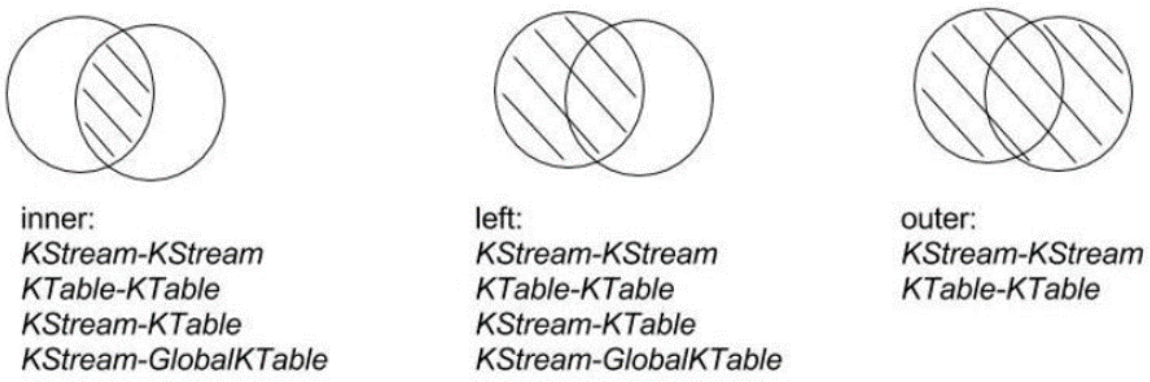


Figure 2.5: Basic type of Joins in Kafka Streams [8]

Kafka Streams Functionalities and Architecture[11]

- Stateful Operations: Kafka has stateless operations as we have seen with filtering or mapping, which don't need to store processing state data. On the other hand, we have the stateful join operation. Also, Kafka Streams has other stateful operations: **Count**, **Reduce**, and **Aggregate**. For these transformations, it is necessary to keep track of previous records, so a state store is necessary to use for Stateful Operations. It works like a backup to ensure fault-tolerance.

- We have analyzed how the Stream DSL works but also Kafka Streams has a Processor API which is a more flexible tool with more capabilities at a low level. It can directly access a state store. Additionally, it can use punctuators to schedule tasks based on either stream time (trigger actions based on record timestamp) or wall-clock time (trigger actions based on actual processing time elapsed).

- Tasks are the basic unit of work and each Kafka Streams instance splits the workload into tasks. How many tasks one application will have is similar to the partitions of the input topic. If a topic has a different number of partitions the number of tasks is set up from the larger number.

- Threads in Kafka increase parallelism and by default, a Kafka streams instance sets up one thread. This value can be increased and is useful for a system if we have topics with more than one partition. As a result, we will have more than one task so each thread can process a specific task. In case we set up more threads than tasks, the extra threads will remain idle.

- Instances work in the same way as threads assigned to tasks but there is a condition: all instances have the same application ID. Adding and removing instances is dynamic, without the need to restart the system(i.e., with zero downtime). In conclusion, each instance can execute in the same computer or a cluster, without limitations.

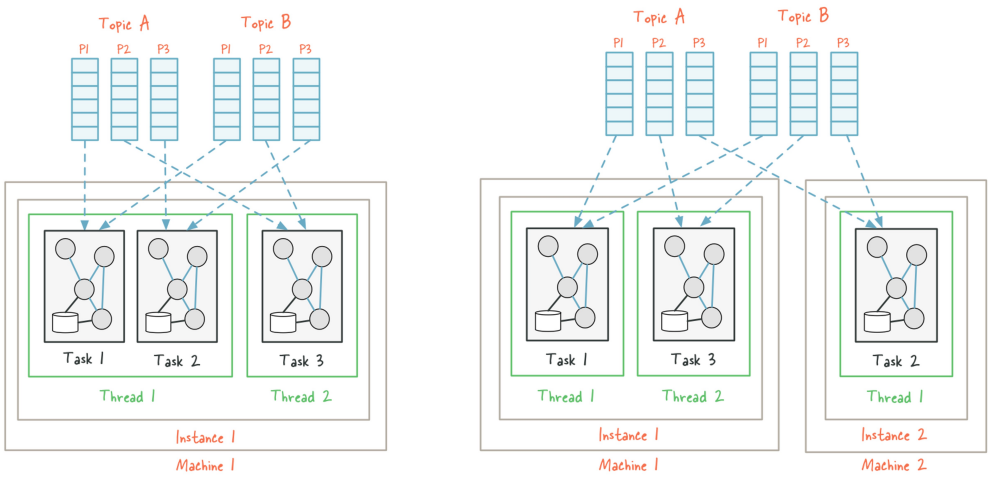


Figure 2.6: A visual Example of how Kafka Streams manage **Topic Partitions**, **Task**, **Threads** and **Instances** [12]

2.3. Data Summarization

By introduction leave space, it is clear fast data summarization [24],[23] and rapid querying capabilities are necessary for efficiently and timely analyzing continuously streaming data. For this reason, a Java class termed Synopses was created to provide a solution to this challenge with an effective representation of large data streams. More specifically, it's an abstract class used for creating new data summarization techniques that extend the Synopses Java class. Here is an example of some of the summaries that inherit from the Synopses class and are supported in SaaMS:

- Sampling methods: They use a subset to represent the whole data stream
- Histograms and Quantiles: These are used to understand the distribution characteristics of the data
- Bloom filter: It's a probabilistic data structure, for instance, verifying the set membership of an element.
- Count Min Sketch: This executes frequency estimation of elements

The Synopses class provides diversity in the summarization using different algorithms based on our needs in stream data analysis with high accuracy and a small error bound which is determined by each algorithm. Adding an algorithm to the Synopses class must fulfill some criteria:

- One pass creation over the streaming data
- Small time required for update
- Estimating queries using summarization
- Capability of merging similar summarizations into one
- Less memory requirements than using raw data

This is the basic terminology of the Synopses class and how it works. Further analysis will be

presented in Chapter 4. In the next section of the thesis, we will do a theoretical analysis of algorithms implemented in the Synopses class.

2.3.1. COUNT-MIN SKETCH

The Count-Min sketch [15] is an algorithm for approximate frequency queries in data streams, providing quick estimation using minimal space. Delving deeply into the theoretical analyses, the Count-Min sketch is a two-dimensional array with parameters (ε, δ), which express the error parameter ε and failure probability δ. In particular, the algorithm guarantees that the frequency estimation error will exceed the error ε with probability at most δ. This algorithm uses a hash function h, which has the purpose of spreading out each element's impact across the sketch to reduce the chance of hash collisions. Also, data is mapped using a d pairwise independent uniformly hash function h via w buckets, with

$$w = \frac{e}{\varepsilon} \quad \text{and} \quad d = \ln\left(\frac{1}{\delta}\right), \text{ as e define the base of natural logarithms}$$

When an update (i_t, c_t) arrives at the sketch, where i_t is the item identifier and c_t is the update value, then c_t is added to each row in the sketch at a specific position, which is determined by the hash function $h_j(i_t)$ for each j from 1 to d. When a query $Q(i)$ of item i arrives, the algorithm calculates the estimation frequency $\hat{a}_i$ as:

$$\hat{a}_i = \min_j \text{count}[j, h_j(i)], \quad \text{constraints: } {a_i}^1 \leq \hat{a}_i \text{ and } \hat{a}_i \leq a_i + \varepsilon \|\mathbf{a}\|_1$$

[1] a_i: defines the actual frequency of item i in the data stream. More specifically, it represents the real count of how many times item i appears in the summary.

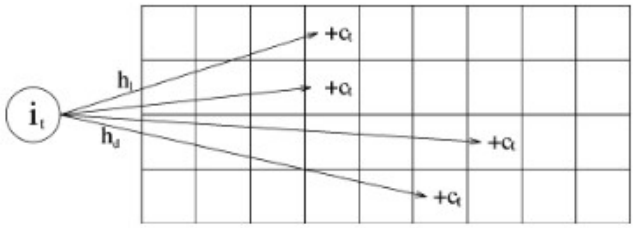


Figure 2.7: Each item i is mapped to one cell in each row of the array of counts: when an update of c_t to the item it arrives, c_t is added to each of these cells. [15]

To review, the count min sketch can be used to solve problems like finding frequencies and heavy hitters in data streams in an efficient way offering a straightforward analysis and improving space and update time compared to other sketches. The space usage in the current sketch is $O\left(\frac{1}{\varepsilon}\right)$ compared to other sketches where the space might grow as $O\left(\frac{1}{\varepsilon^2}\right)$ which is significant development in real applications.

2.3.2. HYPERLOG LOG

The HyperLogLog[20] algorithm is used to estimate distinct count cardinality in large data sets using less memory than direct counting with high accuracy. To better understand how HyperLogLog works, one must study each part of the algorithm, starting with the use of a collection of m registers, where each register stores an integer value. The total number of registers m is determined by the user through the parameter b using the mathematical formula:

$$m = 2^b \text{ with } \mathrm{b}^2 \in \mathbb{Z}_{>0}$$

This algorithm uses a hash function h for transforming each item in Dataset D to the binary domain $D \to [0, 1]$ achieving a uniform distribution of values to accomplish better accuracy in estimations. After this step, it seeks to locate the leftmost $1-bit$ $p(s)$ [3] in the binary representation and will use the $p(s)$ values to update registers $M[1], \ldots, M[m]$. More specifically when an element arrives, the

[2]Based on how much the price b increases, the accuracy increases accordingly but more memory is required and the same applies vise versa.

[3]With s symbolized the hash values of an element

algorithm uses the initial bits of the hashed element v to find which register $M[j]$ will be updated and it also uses the rest of the bits to calculate $p(s)$. This bit is then used for the calculation of the new register value $M[j]$, which is updated with the new $p(s)$ value if it achieves the maximum between the current value and the register $M[j]$:

$$M[j] = \max(M[j], p(s))$$

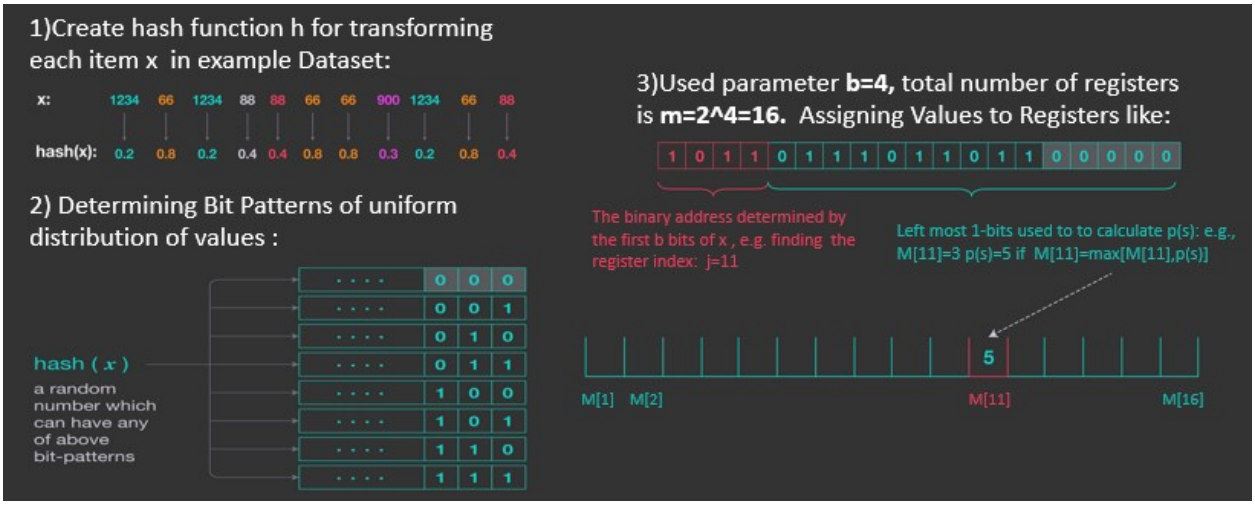


Figure 2.8: Visual representation of the HyperLogLog algorithm [19]

Furthermore, the algorithm computes the indicator function Z, which is expressed as the harmonic mean of the $2^{M[j]}$ values and is calculated by the expression:

$$Z = \left(\sum_{j=1}^{m} 2^{-M[j]} \right)^{-1}$$

The algorithm based on this indicator function Z, estimates the number of unique elements in a set. Also, it applies a normalized factor $a_m \times m^2$ to ensure that the estimated cardinality E is as close to the actual number of unique elements in the set as possible. The final expression of cardinality E is:

$$E = \alpha_m \times m^2 \times Z$$

Where a_m is a constant variable that depends on the number of registers m and m^2 is the square of the number of registers m.

In conclusion, the HyperLogLog algorithm provides a more efficient estimation of the number of unique elements in large-scale datasets compared with the traditional algorithms which require enough memory. The efficiency is confirmed with a small relative error equal to [4] $\pm\frac{1.04}{\sqrt{m}}$ which can be determined according to the tolerance margins of each application.

2.3.3. BLOOM FILTER

The Bloom Filter [7] is a space-efficient probabilistic algorithm that answers the question if an element is a member of a set or not. It can sometimes give a false positive answer but not a false negative for a value. More specifically, it can falsely estimate that a value is a member of a set but not vice versa.

The algorithm uses as its structure a bit array of m bits with initial values set all to 0. The length of the bit array is defined as m, n is the number of expected elements to be inserted, and k is the number of distinct hash functions used. Each of these functions has as its purpose to map an input element onto the bit array. When an element x arrives, it is processed by a total k hash function. Each hash function is denoted as:

$$h_i(x), \quad i = 1, 2, \ldots, k$$

The bit positions in the bit array are represented as:

$$h_1(x), h_2(x), \ldots, h_k(x)$$

For each of these bit positions, the bit is set to 1. It remains unchanged if it's already initialized to 1 from a previous element. Furthermore, the algorithm examines if an element y is in the set by computing k hash functions of this element. The next step is checking if all the bits in the bit array are 1. In case that is true, it's probable that y is in the set, if there are 0 in any of these positions, then it's definitely not in the set.

[4] The $\pm$ defines that the estimation can be either lower or higher than the true cardinality

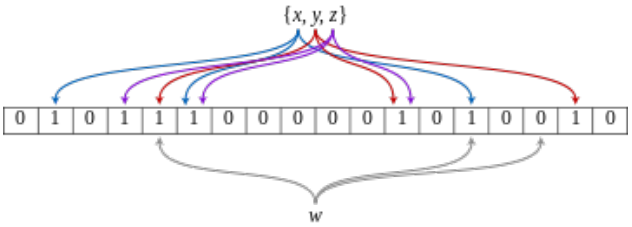


Figure 2.9: the set x, y, z mapped by hash functions to the positions in the bit array. The element w isn't a member of the set because it hashes to one bit-array position containing 0 ($m = 18$ and $k = 3$) [38]

The reason the algorithm doesn't respond with certainty is the limited size of the bit array. As a result, different values may have the same set of positions. The probability that all k positions are 1 for an element and this element didn't appear in the set is:

$$FP = \left(1 - e^{-\frac{kn}{m}}\right)^k$$

To minimize the probability of a false positive FP result, it is necessary to set up the number of hash functions using this formula:

$$k = \frac{m}{n} \ln 2$$

In summary, the bloom filter is a space efficiency algorithm to check membership and produces accurate results with small errors due to its probabilistic nature. The main characteristic that makes it a good choice is that it cannot estimate a false negative.

2.3.4. DFT

The use of Discrete Fourier Transform (DFT) provides a transformation of a sequence of numbers $(x_0, x_1, \ldots, x_{w-1})$ with fixed size w in the time domain, into another sequence of complex numbers $(X_0, X_1, \ldots, X_{w-1})$ in the frequency domain. The representation in the frequency domain of each point X_F contains amplitude and phase. The amplitude is calculated by $|X_F| = \sqrt{a^2 + b^2}$ and the phase by $\theta_F = \arctan\left(\frac{b}{a}\right)$. Based on this the X_F can be represented in two formats $X_F = a + jb$ or $X_F = |X_F|e^{j\theta_F}$. After defining how complex number is represented in the frequency domain, we

can define the DFT coefficients as:

$$X_F = \frac{1}{\sqrt{w}} \sum_{i=0}^{w-1} x_i e^{-2\pi F i/w} \quad F = 0, 1, \ldots, w-1$$

This transformation is used to find across data streams similarities and patterns that cannot be observed in the field of time.

The DFT algorithm [40],[24] uses 3 basic parameters determined by the user. The first one is the basic window size b[5] which defines the range of data time that will be used in each transformation. The second parameter is the sliding window w[6] which represents the total DFT size and contains k number of basic windows. The last one is the number of DFT coefficient n[7] which is used to represent the frequency domain resolution.

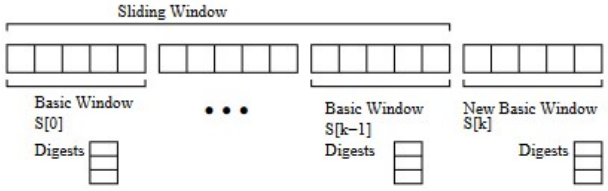


Figure 2.10: Sliding windows and Basic windows representation [40]

The algorithm uses a dynamic data structure for managing the values of the time series. That offers flexibility in the addition and removal of elements that are necessary for a sliding window context. When a new item arrives, it implements a custom logic for updating the appropriate DFT coefficients that the new window adopts instead of calculating all of them from scratch. In particular, the DFT algorithm adjusts the existing coefficients to integrate the new data providing an updated representation in the frequency domain with minimal computation cost. The mathematical

[5]The increases of the b providing greater resolution

[6]When the w increases prove a better smoothens in the data contained in the window, overlooking short-term variations

[7]The size of n determined the compression in an existing resolution, as fewer coefficient digits used as more compressed it is

expression of the m-th DFT coefficient is:

$$X_m^{\text{new}} = e^{\frac{j2\pi mb}{w}} X_m^{\text{old}} + \frac{1}{\sqrt{w}} \left(\sum_{i=0}^{b-1} e^{\frac{j2\pi m(b-i)}{w}} x_{w+i} - \sum_{i=0}^{b-1} e^{\frac{j2\pi m(b-i)}{w}} x_i \right), \quad m = 1, \ldots, n$$

The part of the type $\sum_{i=0}^{b-1} e^{\frac{j2\pi m(b-i)}{w}} x_i$ symbolizes the digits ζ_m and helps to successfully compute the DFT coefficient without recalculating from scratch. Another advantage of this algorithm is the focus on the first n coefficients which represent the most significant frequencies. This logic compresses the data by reducing the dimensions of the data by cutting less significant frequencies. The combination of these two functionalities decreases the requirements for processing and storing.

The DFT coefficients have a range equal to $[-\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}]$ and the feature space $\mathbb{R}^{2n}$ is a cube with a radius $\frac{\sqrt{2}}{2}$. Based on this the algorithm uses a grid structure to estimate near neighbors effectively. It uses the first $\hat{n}$[8] dimensions of the DFT future space for indexing and must apply $\hat{n} \leq 2n$. This feature space defines a $\hat{n}-$dimensional orthogonal regular grid and is partitioned into cells as a result creating a grid with cells of the same size and space. The number of cells in the grid is given by the mathematical formula:

$$\left(2 \left\lceil \sqrt{\frac{\sqrt{2}}{2\epsilon}} \right\rceil \right)^{\hat{n}}, \text{The } \epsilon \text{ defines the cell diameter}$$

The first $\hat{n}$ normalized DFT coefficient is used for mapping the time series as already mentioned, but also for locating neighboring series to identify similarity. More specifically a time series x is hashed into cells $(c_1, c_2, \ldots, c_{\hat{n}})$ and a dynamic threshold is used to compare the correlation coefficient to decide about similarity. If the correlation coefficient is above $1 - \epsilon^2$, a strong correlation exists and the stream is compared with the same cell streams. Otherwise, if the correlation coefficient is lower than $-1 - \epsilon^2$, the algorithm doesn't compare it with the cell into which it is hashed, but with neighboring cells. However, this allows the algorithm to ensure that all streams with a correlation above a certain threshold are compared but this can create false positives, which are filtered out by calculating pairwise distance between the DFT coefficients of the streams.

[8] with $\hat{n}$ defined as the normalized DFT coefficients

In conclusion, the analysis of data in the frequency domain constitutes a common practice to gain useful information about data which would be impossible in the time domain. The most common use of this approach is to identify underlying patterns, offering a powerful tool for processing and analyzing data.

2.3.5. LOSSY COUNTING

The Lossy Counting [27] is a deterministic algorithm that computes frequency counts of elements in data streams in an effective way. The algorithm uses an error parameter ϵ[9] with range $[0, 1]$ which defines the maximum error margins in estimation. Based on this error ϵ it determines the bucket's width which is used to divide the stream. The mathematical expression of this is:

$$w = \left\lceil \frac{1}{\epsilon} \right\rceil$$

Each bucket is labeled with bucket IDs that have an initial value equal to 1. The current bucket id denoted as $b_current$ and calculated by:

$$b_{current} = \left\lceil \frac{N}{w} \right\rceil, \text{where N denotes the current length of the stream}$$

The algorithm uses a data structure D which is represented as a set of entries like this (e, f, Δ), where e is a stream element, f is an integer representing estimated frequency, and Δ is the maximum possible error in the estimation of frequencies f. When a new item e arrives first, it searches in data structure D to determine if e already exists. In the case that this element is detected, it increases frequency f by one, otherwise, it creates a new entry in the data structure like this $(e, 1, b_{\text{current}} - 1)$. To ensure that memory usage remains bounded, it uses a compression technique that doesn't affect the algorithm's accuracy. The rule for deleting an entry (e, f, Δ) to achieve compression is defined by the inequality:

$$f + \Delta \leq b_{current}$$

[9] When the error ϵ is close to 0, it estimates with better accuracy but needs more memory

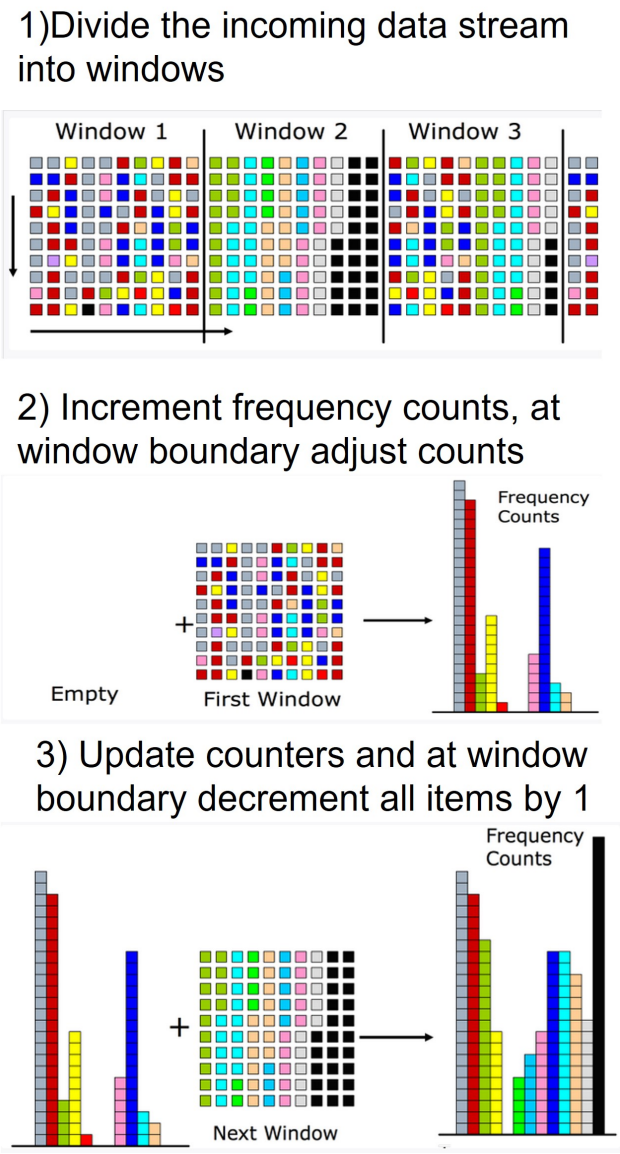


Figure 2.11: Visual representation of the Lossy Counting algorithm [29]

Additionally, the algorithm can provide an estimation S with a list of the most frequent items for a specific threshold s for selecting which elements must be in the list. The formula for this selection is denoted as:

$$f \geq (s - \epsilon) \times N$$

Besides this, it can also provide an estimation count of frequency f for each item in the data

structure. This count is the number of times a specific item e has been seen since it was added. This value has a deviation based on the value Δ which is assigned when a new entry arrives and remains unchanged.

Finally, the Lossy Counting algorithm never produces false negative results but the number of false positives is limited. Making it a good approach for stream processing analysis where direct counting is inefficient due to memory usage.

2.3.6. STICKY SAMPLING

The Sticky Sampling [27] is a probabilistic random sampling algorithm for computing $\varepsilon-$ deficient synopsis in a data stream, ideal for handling large or infinite data streams effectively. This algorithm has 3 parameters which take their values from the user. The first parameter is support (s) which is defined as the bound in the stream at which an item is characterized as "frequent". The second parameter is error (ϵ), which is used to define the error level of an item that is acceptably deviated from its true frequency. Finally, the parameter (δ) is the probability of failure, which indicates the likelihood that the algorithm will fail in providing an accurate estimation inside the error bounds ϵ. In addition to the parameters defined by the user, there is another parameter, the sampling rate r which is initialized as $r = 1$ and describes how elements in the data stream are sampled. This algorithm also has a data structure S which stores the collection of entries (e, f) where each entry has elements e from the data streams and estimated frequencies f.

All the new elements arriving at the data structure if they don't exist are added to the structure as $(e, 1)$ with a probability $\frac{1}{r}$, otherwise, the frequency f is increased. This means that the likelihood of a new element being added to the data structure depends on $\frac{1}{r}$. Furthermore, the sampling rate r varies over the lifetime of a stream, let

$$t = \frac{1}{\varepsilon} \log(s^{-1}\delta^{-1})$$

The first $2t$ elements are sampled with rate $r = 1$ the next $2t$ are sampled with $r = 2$ and the next

$4t$ are sampled with $r = 4$. The simple rate r is a geometric progression:

$$r = 2^i, \text{ with } i = 0 \ldots \infty$$

Each time the sampling rate r changes the algorithm updates the frequencies in the data structure S. In addition, for each entry (e, f) repeatedly tossing an unbiased coin until a success is obtained for every failure, it decreases the frequency f by 1. If before success is obtained the frequency f becomes 0 the element e is removed from S keeping only important items. This compression functionality has as a result decreased memory needs without affecting algorithm accuracy. The number of unsuccessful coin tosses follows also a geometric distribution.

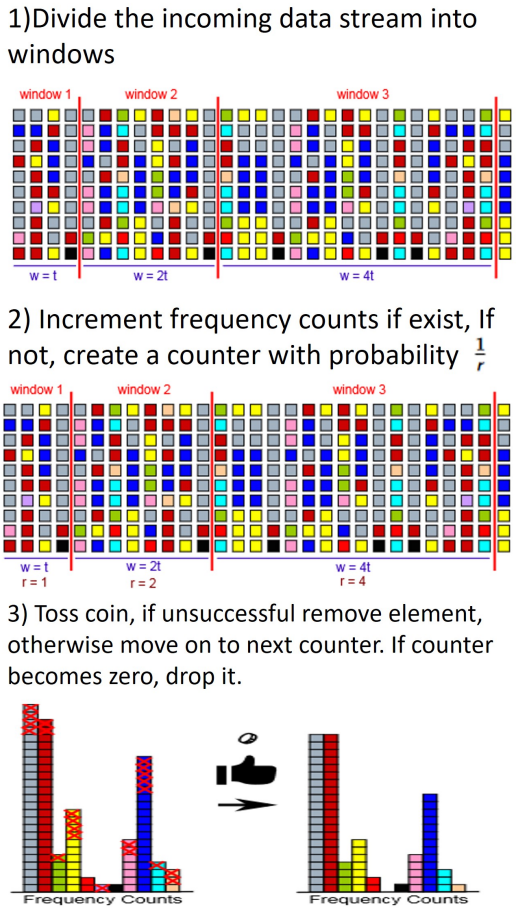


Figure 2.12: Visual representation of the Sticky Sampling algorithm [18]

Furthermore, the algorithm can provide an estimation of a list of elements when it is requested for a specific threshold s for selecting which elements must be in the list. The formula for selection is denoted as previously in Section 2.3.5:

$$f \geq (s - \epsilon) \times N$$

Also can provide an estimation frequency for a specific element in the data structure besides the ϵ-deficient synopsis which was described previously.

In summary, the Sticky Sampling algorithm has efficient space complexity making it suitable for processing large stream data with little use of memory. It proves by the computation of the ϵ-deficient synopsis with probability $1 - \delta$ and uses at most $\frac{2}{\varepsilon} \log(s^{-1}\delta^{-1})$ number of entries.

2.3.7. AMS SKETCH

The AMS sketch [1] is a space-efficient probabilistic algorithm that estimates the frequency moments of a data stream. The frequency moment F_k[10] represents important demographic information about the data and is used for extracting important statistics. As the frequency moment F_k is defined as:

$$F_k = \sum_{i=1}^{n} m_i^k$$

Where m_i represents the number of occurrences of i in the sequence A[11] and n is the range of distinct possible values each element a_i can take from the set N[12]

This algorithm uses two parameters: depth s_1 and width s_2. The first parameter s_1 sets up the number of pairwise-independent hash functions. The second parameter s_2 determines the number of counters in each layer of the sketch. The product of $s_1 \times s_2$ produces the total number of counters tot_{count} and defines the total size of the sketch matrix that the algorithm uses as a data structure.

[10] with k representing the k-th power of the frequency moment F_k

[11] A is a sequence of elements $A = (a_1, a_2, \ldots, a_m)$

[12] As N denotes the set $N = \{1, 2, \ldots, n\}$ which defines the range of distinct values that elements of set A can be assigned

Each time a new element a_m arrives from the stream, the algorithm decides with probability $\frac{1}{m}$[13] if the new element a_m will replace the current a_l element. In the case where a_l is replaced by a_m, then the counter r [14] must be initialized to 1. Otherwise, if the a_l doesn't get replaced, there are two cases: the first one is a_m equal to the a_l, then the r is incremented by 1 and the second one is the a_m not equal to the a_l, then the r remains unchanged. Additionally, for each processed element that is added to the sketch, there is a random variable X with the expression:

$$X = m(r^k - (r-1)^k),$$

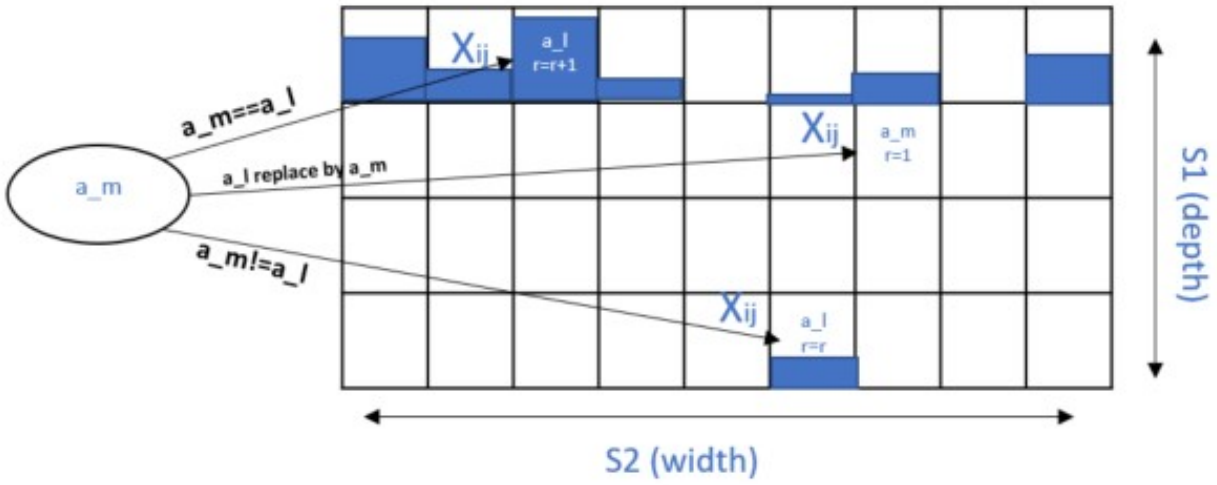


Figure 2.13: Visual representation of the AMS Sketch algorithm

The algorithm estimates the F_k frequency moment in the sketch by calculating firstly the random variable Y_i. Each Y_i is the average of s_1 random variables X_{ij}, where $1 \leq j \leq s1$. Especially, estimating the second frequency moment F_2 requires calculating the median of all Y_i values. The use of the median ensures that the estimation is less sensitive to extreme values. Finally, the estimation is proven to have an expected value equal to the true frequency:

$$E(X) = E(Y_i) = F_k$$

[13]With m defining the index of the currently processed element a_m in the stream
[14]with r symbolizing the number of occurrences for a selected element

Also, the improvement in the accuracy of the estimation is secured by:

$$Var(Y_i) = \frac{Var(X)}{s_1}, \text{ where}$$

$$Var(X) = E(X^2) - E^2(X)$$

To sum up, AMS Sketch is an algorithm for data stream analysis with a good approach to ensuring accurate results despite its probabilistic nature. It provides a better calculation result due to the use of F_2 frequency moment for estimation, which achieves a deeper analysis of data.

2.3.8. GK QUANTILES

The Greenwald-Khanna Quantiles algorithm [21] provides a design for efficient estimation space-efficient computation of quantile summaries of very large data sets in a single pass without requiring knowledge about the size of data sets.

It uses the parameter epsilon ϵ[15] which denotes the precision of quantile estimates. The data structure S of the algorithm is a dynamic set of tuples and each tuple is defined as:

$$S = (u_i, g_i, \Delta_i)$$

where i indexes the tuples, u_i is the value of the i^{th} tuple, g_i is the number of items similar to u_i and finally Δ_i represents the range error.

Each summary must always include the minimum and maximum elements u_0 andu_{s-1} which have been calculated so far, so when a new item x arrives, if it is the minimum or maximum, or x is the first added element, it is inserted as $(x, 1, 0)$. Otherwise, it is inserted as $(x, 1, \Delta)$, with the parameter Δ[16] being calculated based on x position relative to the other elements in the summary. Additionally, the algorithm executes a compression of the summary with the merging of suitable tuples

[15]The ϵ has a range $[0, 1]$ and lower it is, more space is required

[16]The Δ value has an important role, ensuring that the epsilon error is maintained bounded

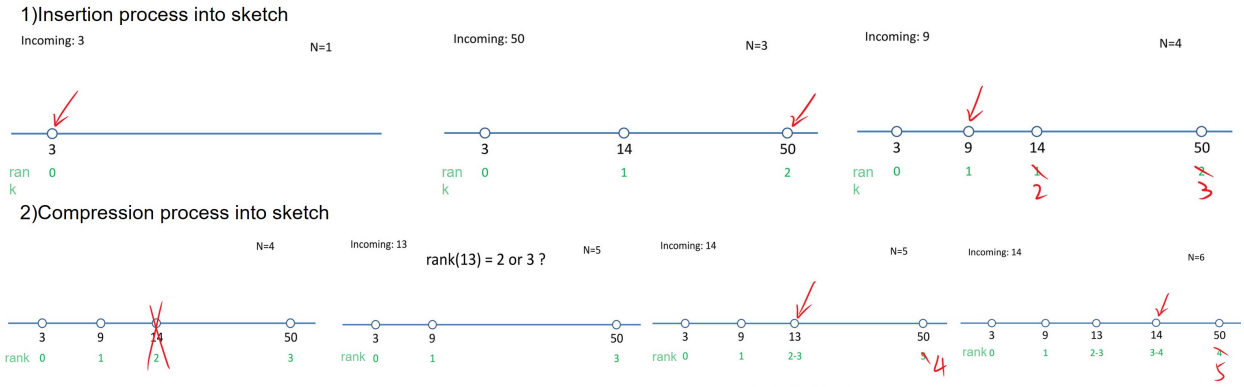


Figure 2.14: Visual representation of the GK Quantiles algorithm [37]

Estimating a quantile $\hat{Q}$ with an acceptable range of value$0 \leq q \leq 1$. First needs calculating the rank R:

$$R = \lceil q \times N \rceil$$

Where N denotes the total number of elements processed by the algorithm. After that, the algorithm goes across S and sums the g_i values of each tuple and denotes it as $G_{totalsum}$. This process is repeated until the smallest value of u_i is found that applies $G_{totalsum} \geq R$. The mathematical expression of this is:

$$\hat{Q} = \min\{u_i \mid G_{\text{totalsum}} \geq R\}.$$

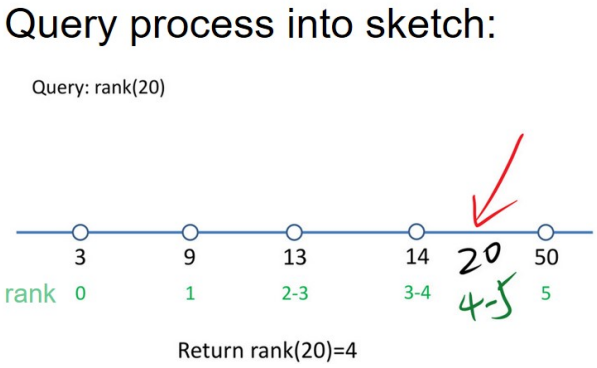


Figure 2.15: The estimation process of the GK Quantiles algorithm [37]

The GK Quantiles algorithm is a perfect solution for large data streams with $O\left(\frac{1}{\epsilon}\log N\right)$ complexity

and it provides a significant development compared to the best existing similar algorithms, which have space complexity $O\left(\frac{1}{\epsilon}\log^2 N\right)$.

CHAPTER 3

DESIGN OF THE SYNOPSES AS A MICROSERVICE

In this section, we look into the important functionalities and operations that were dominant in the development of Synopses as a Microservice (SaaMS), which leverages Kafka and Kafka Streams. The general characteristics of our system are described by non-functional requirements, including system performance for scalability, durability, and other attributes that ensure efficiency and reliability in the system. On the other hand, the term "functional requirements" refers to data processing, stream management, and capabilities for real-time analysis.

The purpose of our thesis is based on the instantaneous and continuous analysis of vast data streams using Kafka and Kafka Streams. This analysis focuses on monitoring financial markets to detect patterns that can be used for investment opportunities as an exemplary use case. However, the possibilities of SaaMS are not limited only to financial data analysis but can be expanded to other fields like medical and agriculture applications. In general, SaaMS can be applied anywhere where processing data at high velocities and controllable accuracy in real-time is necessary.

3.1. Non-Functional Requirements

SaaMS has an architecture that can serve a large scale of applications that need answers in real-time. It can work autonomously or in cooperation with other more complicated architectures as a subset. Independently of the case in which it is used, it must take into account some constraints to ensure consistency in the services it provides. In the next section, further analysis of how SaaMS ensures high-quality services will be provided.

3.1.1. MicroService Architecture

In our context, the microservice architecture combined with Kafka and Kafka Streams to address the specific needs of real-time data processing. We have already discussed in Section 2.2 about Kafka and Kafka streams and their ability to process data with high throughput on the fly. In this section, we focus on the microservice architecture and how it ensures scalability, flexibility, and maintainability in stream processing. The main purpose of using microservices like Router

and Synopses in our case is that each one manages its data and state. This ensures the necessary autonomy across microservices and protects against failures. For example, if a synopsis fails, the other synopses will continue to work without being affected at all. This is due to the architecture of SaaMS which is presented in the next figure:

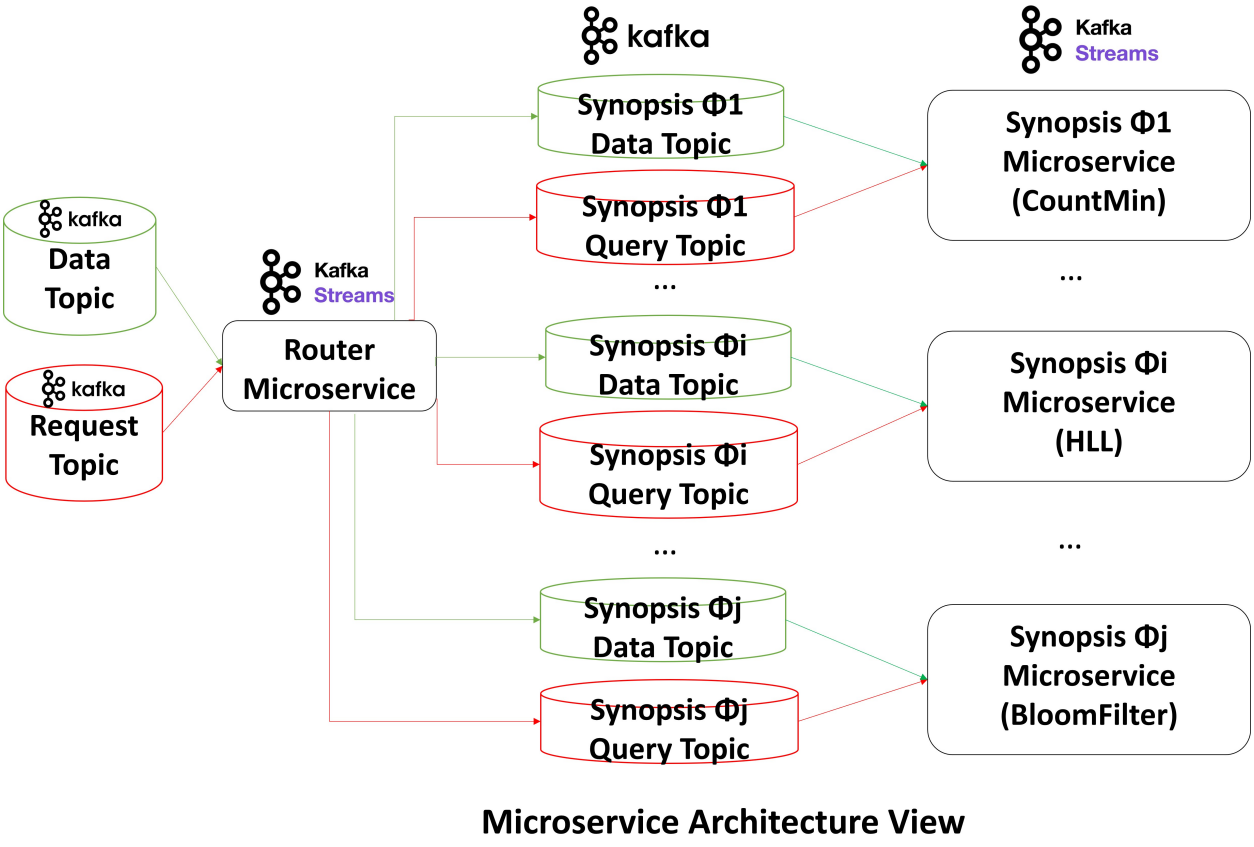


Figure 3.1: SaaMS architecture

The key point of this architecture is the use of different Kafka topics for each microservice, ensuring the independence of each one. More specifically looking into the analysis of the Router microservice, it is clear that it has the role of a manager. It can be characterized as an intermediate step that is responsible for the dynamic building of Synopses microservices and for the correct routing of queried and new streaming data tuples to them based on a specific key (StreamID, DatasetKey, SynopsisID, field).

As illustrated from Figure 3.1 the Router contains two Kafka topics Data and Request which are the connection between SaaMS and the external sources that produce these values for the application. Also, each microservice of Synopsis has its topic for data and requests to ensure their autonomy and independence. However, the application doesn't interact with the external sources only by receiving data but produces the results of each algorithm to external outputs Kafka topics. Finally, this architecture uses Kafka and Kafka Streams effectively to produce continuous summarization results for data that arrive from outside in real-time.

3.1.2. Scalability

SaaMS scalability is divided into 3 levels. The first level is horizontal scalability, which represents the system's capability to manage a large volume of data with high velocity. The second level related to vertical scalability, refers to the system's ability to scale with the number of processed streams. The final level is federated scalability, which is about the expansion of a system across multiple distributed environments that are often geographically dispersed.

Further analysis is necessary to clarify how SaaMs utilize these types of scalabilities. To begin with, horizontal scalability is achieved through Kafka which has complicated mechanisms for distributing the workload. The main way to accomplish that is by using more partitions in Kafka topics which increases the parallelism. As a result, it succeeds in maximizing the throughput and minimizing the latency. Also, SaaMS uses Kafka Streams for real-time stream processing, assisting the horizontal scalability more than using other frameworks. There is an analytical description in this paper [17]. The key point is that Kafka Streams is the best choice for an immediate data handling framework. This was evaluated using benchmarks between Kafka Streams and frameworks like Apache Flink, used for managing unsteady workloads, Spark Streaming, and Structured Streaming, used for applications that need maximum throughput and where an increase in latency isn't crucial. This comparison confirmed Kafka Streams as the most suitable framework for direct processing with high horizontal scalability.

On the other hand, the vertical scalability in SaaMS is implemented with the use of Synopses in Section 2.3. The Synopses are algorithms that have been implemented to calculate complex

estimation processes in data streams effectively. More specifically, the design of these algorithms allows tasks that previously required high computational and space costs but now be executed with fewer resources in a short time, scaling up the vertical scalability and saving significant resources.

Additionally, SaaMS exploits federated scalability, the ability to process data in different virtual environments and build up Synopses in different instances that are located in different VMs. This capability decentralizes the data processing and allows Synopses to split their calculations into small parts in different VMs, as a result, using less bandwidth than transmitting all the Synopses.

3.1.3. Pluggability & Portability

The pluggability and portability are abilities of SaaMS application to integrate with components from different sources and to execute in different environments. Based on pluggability requirements our application makes it with the cooperation of Kafka and Kafka Streams. The use of these two frameworks composes the procedure of consuming data from an input topic and producing results for an output topic easily due to the last one is a part of the Kafka ecosystem. So, this makes SaaMS able to process data by sending it to a Kafka topic without any necessary adjustment or use of external frameworks. Furthermore, SaaMS achieves portability by encapsulating the application in a JAR file, making it portable across different environments with the only requirement being a Java-enabled device

3.1.4. Extensibility & Usability

In this section, we talk about how SaaMS achieves Extesiblility and Usability. To begin with, the Synopses class has been implemented using two main object-oriented programming characteristics of Java: polymorphism and inheritance. This gives the ability to easily and quickly extend this class with new algorithms, as has already been described in Section 2.3. Also, each synopsis can be constructed with specific parameters, suitable for each algorithm we want and for the particular dataset. As an outcome, the Synopses class is the key pillar to achieve extensibility in the application. This architecture permits the dynamic addition and removal of instances at runtime to decrease computation load and increase performance. On the other hand, the usability of SaaMS guarantees how it interacts with the user. The only requirement to build and query for a Synopsis

is the use of a simple Request Section 4.1.2.

3.1.5. Reliability

Reliability is an important condition for each system and is split into availability and fault-tolerance. In our application, availability is ensured by the design of Synopses to need short response times. Additionally, the fault tolerance characteristics of Kafka and Kafka Streams reduces the probability of SaaMS having a failure. In the case where a Synopsis fails, the application is designed in that way to continue working. More specifically, the rest of the Synopses will continue to produce the necessary results without being affected. Confirmation of the previous claims is the experimental results of Chapter 5, which show the behavior of the application in different scenarios.

3.2. Functional Requirements

In the previous section, the presentation of the non-functional requirements of SaaMS application was based on the capabilities of the framework that it uses, Kafka and Kafka Streams. Now, it is necessary to introduce the functional requirements that must be implemented by an application during the workflow. These include the ability to collect and process data in real-time from different sources build and query Synopses. The analysis which follows will contain all the functions used in the pipeline to produce the final results.

3.2.1. Single-Stream Synopsis Maintenance

With the term Single-Stream Synopsis, we refer to a compact representation of a data stream with the capability of dynamic update at any time. The summarized data in this case are initialized and updated from only one source sequence. An example of a better understanding is a stock market that contains stocks. The continuous changes in prices of one or more stocks arrive at the input topic from one source and are processed sequentially to produce the necessary analysis.

3.2.2. Multi-Stream Synopsis Maintenance

Besides the previous stream synopsis, there is also a Multi-Stream Synopsis. From the name, it is clear that this type of synopsis is used for processing data from different sources. It isn't necessary that these sources contain different types of data but can use this type of synopses to decrease computation time and cost than using a single stream synopsis. In both cases of Multi-Stream

Synopsis, it is necessary to aggregate the final results from each source to produce the correct final result that will contain the total summary. Examining the same paradigm as previously, the difference in this instance is that we have more than one source from the same stock or different stocks. This changes the sequence characteristic, so the processing now is parallel producing different analyses from each source which must be aggregated to have a total view.

3.2.3. Dataset Synopsis Maintenance

This type of synopsis can be implemented as a single or Multi-Stream Synopsis and provide a total view of a dataset. As a dataset, we refer to a group of many streams(stocks based on previous examples), containing the context of a compact format of the total dataset. For the stock market example, the dataset represents the whole market's stocks and is used to identify a general behavior or trend of the market.

3.2.4. Ad-hoc Querying Request

All the previous categories of synopses must have the ability to answer queries about the current summarization of a synopsis. The main characteristic of these queries is that the result they provide is temporary for the current instance and does not adopt any updates.

3.2.5. Continuous Querying Request

On the other hand, a Continuous query provides the same functionality as the previous request with an important difference. Each time a new value arrives, it updates the previous answer with the new calculation using the whole Synopsis.

3.2.6. Save & Load a Synopsis

Each maintained Synopsis is stored in a specific location in a disc periodically. In the case of Multi-Stream Synopsis stored every time the last merged value is updated. This saved Synopsis can be used to be loaded into the system when necessary. The Loaded Synopsis can be any of the previous types of Synopsis (e.g. Single-Stream, Multi-Stream, Dataset) with the only difference being that it already contains summarized data. Finally, the Loaded Synopsis can be updated with a new tuple of data providing total summaries that combine the loaded and newly arrived data.

CHAPTER 4

IMPLEMENTATION

In this chapter, we will continue with the analysis of SaaMS as in the previous chapter but we will deepen how it works, describing each step analytically. Additionally, in this chapter, we will also present how it is constructed in the Java Synopsis class based on previous theoretical analyses to have an overall view. Moreover in this chapter, we will explain the user interface that is necessary for the interaction with SaaMS application. In the final section of this chapter, we will compare the SaaMS against SDEaaS which is a valuable application for stream processing in real-time built-in Flink.

4.1. Implementation of Microservice Architecture

As outlined in Section 3.1.1 SaaMS application adopts a microservice architecture. The key points of this implementation have already been described but this is not enough to fully understand the functionality when requests and data tuples arrive at our application. For better assimilation of the implementation method of SaaMS, Figure 4.1 was created, which will be analyzed in the following subsections.

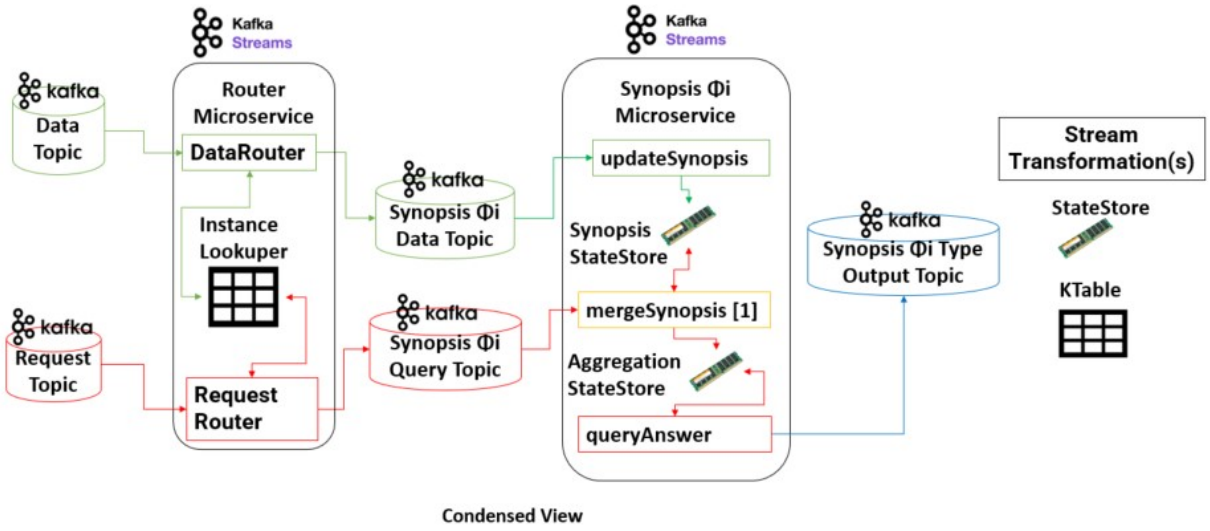


Figure 4.1: The Condensed View of SaaMS architecture

4.1.1. Requesting New Synopsis

When the Request topic receives a Create Synopses request the application based on Figure 4.1 will follow the red line path and handle the following process:

1. In the leftmost part of the architecture, the Router Microservice checks if the data and request topic at the second level (middle of Figure 4.1) Kafka Streams do not exist, creates them and configures partitions based on the desired parallelization degree as declared in the request. In case these topics already exist due to a previous CreateSynopsis request of any type, this step is skipped to avoid unnecessary data duplication.

2. If the synopsis is not already maintained each CreateSynopsis request spawns a new microservice for that synopsis, with a parallelization degree equivalent to the desired parallelization degree as declared in the CreateSynopsis request.

3. Each new synopsis is logged using its type, input topic as a key into a KTable, and is used for checking if a synopsis is already maintained or not. The method it uses to confirm the existence of a synopsis is to group the data based on that specific key. This ensures that it gives a correct answer about maintenance or not.

4. The new Synopsis Microservice is created to handle a specific synopsis algorithm. The first step after establishing the microservice is to create and configure with specific parameters a state store that is used to cache the current state of the synopsis. For instance, in the case of a Bloom filter, the state of the synopsis is its current bitmap representation.

A Create Synopsis request must be implemented based on synopsis types from Section 3.2. Each of these follows generally the same processing logic with minor differences as mentioned above. In the case of single and Multi-Stream Synopses, both types use the same primary key as a key-value pair for the KTable which maintains the Synopsis, but with a different parallelization degree. The key is defined by the **StreamID** (item name),**DatasetKey** (source name),**SynopsisID** (type of summarization), and **field** (type of values summarization established). As for the Dataset Synopsis,

it also implements the logic of the Create Synopsis request but this type has a difference in the primary key used for the KTable. The key is defined as previously mentioned, with the distinction that the **Stream ID** is empty because the DataSet Synopsis is established for the whole data set. Further analysis of the Multi-Stream Synopsis will follow in the section on the parallelization schema.

4.1.2. Query Request

When a Query Request is received, it executes the following operations:

1. In the Router Microservice the only process executed is routing the request to the maintained synopses Request Topic (Second Level) because the actual purpose of this type of request is to offer an estimation for the maintained synopsis.

2. In the Synopses microservice there are two cases depending on the type of the query request as we have seen in Sections 3.2.4 & 3.2.5. For an Ad-hoc request, the unique job implemented is to find the current state (AggregateStateStore) of the synopsis and estimate the requested value. On the other hand, a Continuous request also finds the current state (AggregateStateStore) of the synopsis, but additionally updates the state of the synopsis providing an estimation with each update of the synopsis. In both cases, the estimation values are written in an Output Topic as the blue path of Figure 4.1 shows us.

4.1.3. Updating the Synopsis

In particular, when a new streaming data tuple is ingested, the Router Microservice looks up the InstanceLookuper Ktable for the keys to define to which Data Topics of each maintained synopses the tuple should be directed. Sometimes, the data tuples that are arriving may not match with any maintained synopses. In this case, the tuple remains in the Data Topic for a predetermined time until a synopsis is established to ingest it. Every Synopsis Microservice has a StateStore that is used to store the current state of the synopses and is initialized with request parameters, as we already discussed. As a result, when data tuples arrive from the second-level data topic, they use the add operation that the synopses have to update them in the StateStore. If there is a Multi-

Stream or Dataset Synopsis, the StateStore is local to each instance that maintains the synopsis. As an outcome, a merge operator is needed to export the global summarization, more about this operation will be analyzed in the parallelization schema. The total view of a synopsis for all types is provided by a Synopses KTable which is on standby if a Query request arrives to perform the overall estimation and writes this result in an output topic. There is a single output topic for each type of synopsis, i.e., all maintained CountMin sketches output to the same topic, and the results are distinguished based on the **synopsesID** key, i.e., the output topic is not bound to the Microservice but to the synopsis type. Also, it is created the first time we handle a request to maintain that type of synopsis, e.g. the first time a countMin sketch is requested.

The Router Microservice includes a series of transformations for handling input data from the Data Topic as well as a separate series of transformations to handle application requests. The RequestRouter transformation includes:

- A **FlatMap** transformation to transform the Request list (which may include one or more requests submitted simultaneously) into individual requests
- A **Group by** transformation to ensure that we don't recreate already maintained synopses as already represented in a previous section
- A **MapValues** transformation to send requests to the requested topic of each Synopsis Microservice of the second level and to instantiate the microservice if it does not exist, following the red path in the Figure 4.1

On the other hand, The DataRouter transformations represent the green path in Figure 4.1 and include the necessary processing with **repartitioning Round Robin** [17] based on whether the maintained synopsis is for the whole dataset or in a specific stream. Also, there is a category for no mergeable synopses which is established in the whole dataset and it must be **partitioned based on StreamID**. The DataRouter transformation implemented includes:

[17]Round Robin algorithm is used to equally and fixedly allocate data. It assigns data to slices to each process in equal portions and circular order, handling all processes without a priority algorithm.

- A **Repartioning** of the data on a case-by-case basis.

- A **KStream-KTable Join** to join the tuples from the data topic (first level) with the InstanceLookuper Ktable info to check to which data topics of the second level this tuple should be routed to

In addition to the Router and Synopsis Microservices, there are transformations for handling data and requests that arrive at topics from the second level. Among the transformers that are materialized are:

- Different **Transform** transformations each one having a specific job like initializing the state store based on the Create Request parameter. Also, a Transform uses a punctuator with a static time value to batch the data that has arrived from the second level Data topic before implementing the add operation for updating the synopsis to the local StateStore. This provides a better performance than adding individual data tuples to the StateStore. Eventually, another Transform for the Multi-Stream and Dataset Synopsis is used to fix the problem of random arrivals[18] with each StateStore instance writing to another Aggregate StateStore using its instance number(partitionNumber) as a key. The correct values are recovered from the Aggregate StateStore and merged to produce the final Result.

- A **KStream-KTable Join** to join the final synopsis state for both parallel and non-parallel cases with the query requests for capturing the current state of the synopsis.

- A **MapValues** transformation to print in the screen and write to the output topic the estimated value that has been requested.

[18]In parallel processing data doesn't process in sequence creating synchronization problems. It is important to solve this inconsistency for the correctness of each application

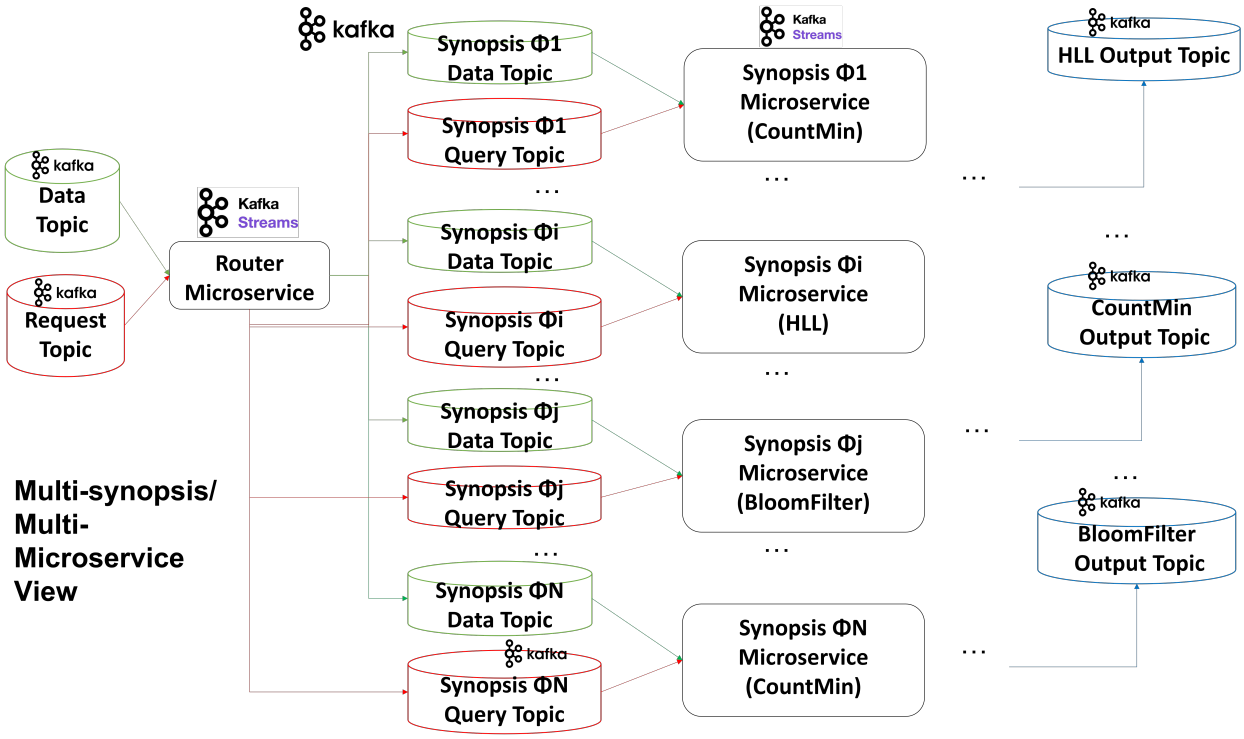


Figure 4.2: SaaMS architecture using a Multi-Synopsis and Microservice representation

The basic functionalities of SaaMS architecture as represented in Figure 4.1 have been expanded extensively. Figure 4.1 constitutes a condensed view of this program and how it behaves for one Synopsis. A more multidimensional representation of a SaaMS with more microservices depicted is necessary for understanding the whole functionality. For this reason Figure 4.2 was created to represent the structure of Synopsis Φ_i Topics and Microservices in SaaMS application. More specifically, each requested synopsis has its microservice and two Kafka topics (Request and Data) to implement its functionalities independently from other Synopses. It is clear from Figure 4.2 that we can maintain the same type of synopsis e.g. CountMin, multiple times but it is required to be established with different parameters (StreamID,DatasetKey,SynopsisID,field). Another observation that can be extracted from Figure 4.2 is that summaries of the same type use the same Output Kafka Topic.

4.1.4. Parallelization Schema

In the previous section, we detailed both descriptively and visually SaaMS implementation architecture. In this section is important to concentrate on the processing capabilities of the application working in parallel. This functionality is important for success increasing throughput and decreasing latency. Also, it is clear from Section 2.2 where we analyzed Kafka and Kafka Streams services, that based on these frameworks, we can implement parallelization in applications. In the same way, SaaMS uses Kafka and Kafka Streams to implement a Parallelization Schema.

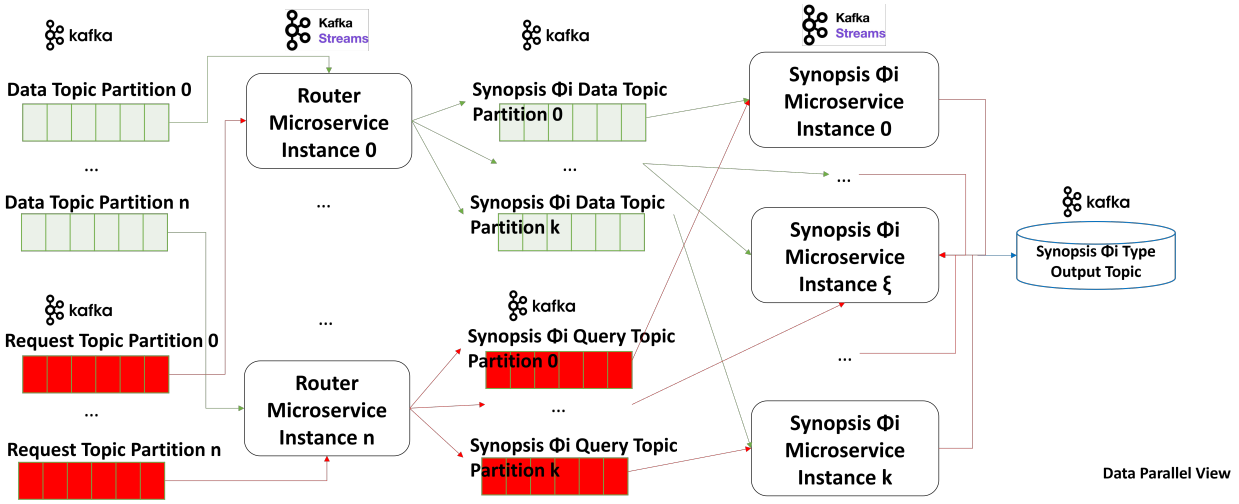


Figure 4.3: Parallelization Schema in SaaMS architecture

In Figure 4.3 the parallelization model for SaaMS application is represented. It is noticeable that parallelism is implemented at two different independent levels: Router and Synopsis level.

To begin with, the Router level has a static number of parallelism which is determined by the user when the program starts. More specifically, based on this number, the number of partitions that the Data and Request topic will have is determined. For example, if the user defines parallelism as equal to four both topics will have 4 partitions. In the Data topic, data is partitioned based on streamID and DatasetKey using the Round Robin algorithm. On the contrary, the Request topic also has the same number of partitions as the Data topic but it uses only the first, the other partitions are IDLE until new Instances of Router Microservices are added. The number of partitions in this topic

is the upper bound of Router Microservice instances. All the stateful operations like join, group, etc., which have a StateStore for intermediate values adopt the same number of partitions for these StateStores, as the Request and Data topic. This functionality is implemented automatically by the Kafka Streams API.

However, not only the Router Microservice works in parallel but also the Synopses microservice. The parallelization in these microservices is determined by the Create Synopses request, as a result, each Synopses microservice can use a specific parallelization degree based on the workload of summarization to provide scaling. Also, every Synopsis uses this degree of parallelization to define the number of partitions in second-level topics. The Φ_i Request topic contains in its first partition the original request which is routed from the Router microservice and replicas of this request in the other partitions. This technique ensures that k local Synopsis StateStores will be created like the number of partitions. As a result, each one will be initialized from the same parameters and will host a part of the Synopsis allowing managing in parallel streams from different sources. In the Synopsis level the Φ_i Data topic and StateStores of the stateful operation work in parallel as in the previous router level. Parallel processing is usually more efficient because it splits the procedure into smaller parts but it makes it necessary to concentrate on and sum up sub-results. For this reason, it is necessary to merge StateStores partitions to have the whole picture of a Synopsis.

The previous Parallelization Schema can be achieved in SaaMS in two ways, using one instance (one VM) with multiple threads or using more than one instance (multiple VMs). Also, it is possible a combination of both, meaning multiple threads running each in one VM and we can have multiple VMs. The first one is the most simple method to implement parallelism and defines the number of threads which is equivalent to the parallelization degree of the application. In the context of fewer threads than partitions, this means that in one thread more than one partition is assigned, otherwise the extra thread remains idle. The second type of parallelism is using more than one instance combined with threads. It is a more complicated process to add extra resources in SaaMS than to use one instance with multiple threads. This functionality becomes mandatory to use an extra topic in the Router Microservice with one partition for managing instances. When an

instance of Router Microservice begins first of all, it sends a report on this topic about the count of currently active instances. After that, only one of the active instances forwards a replica to the next partition of current requests to start other instances. This replica is responsible for starting the new instances of Synopsis Microservice. After that, Kafka will take care of distributing and balancing the processing logic across instances.

4.1.5. Elastic Scaling to multiple VMs

In this section, we will analyze further how Kafka and Kafka Streams capabilities are used in cooperation with SaaMS to manage parallelism and elasticity. The term **"Instances"** can easily create confusion due to the use of the same term for the Kafka Streams Instances and Router or Synopsis instances. To clarify, the last refers to Java classes that are built on top of Kafka Streams and encapsulate distinct processing logic of this framework. Therefore each instance of Router or Synopsis Microservice contains a Kafka Streams instance that implements the necessary processing logic that has been explained in previous sections. After these required definitions we will explain how SaaMS can work in parallel with adding or removing instances dynamically when working in a distributed environment with multiple VMs.

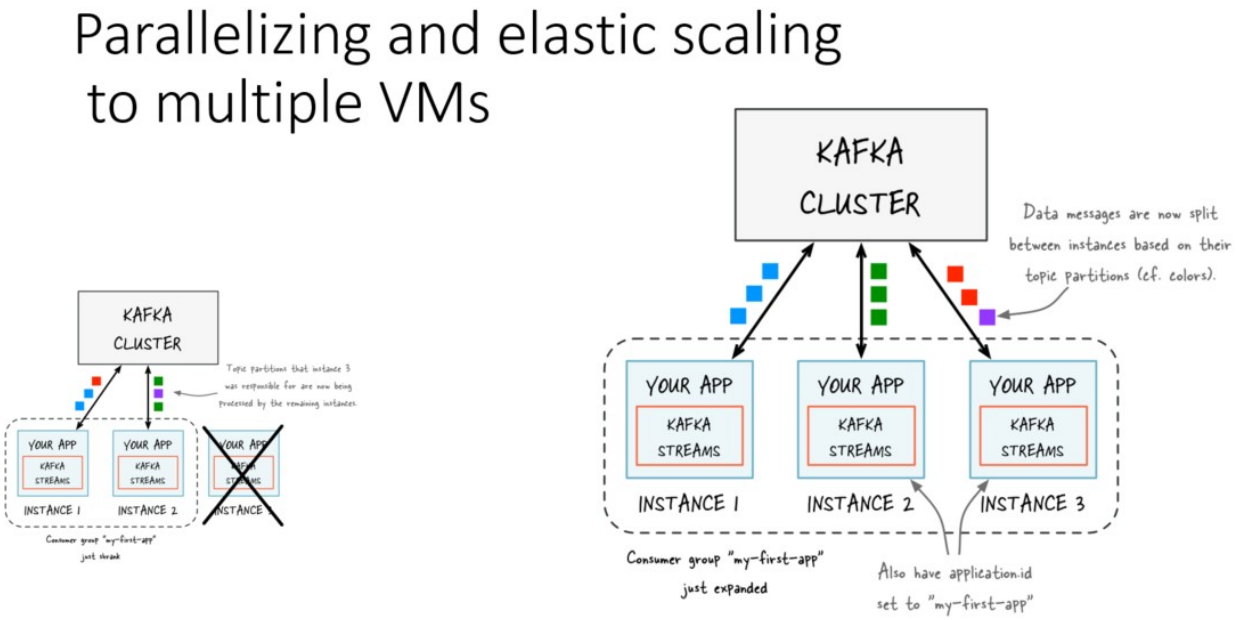


Figure 4.4: Adding and Removing instances of SaaMS dynamically

The use of multiple instances can increase or decrease the processing power of the application due to the characteristic that Kafka Streams has for balancing the workload among instances(VMs). Figure 4.4 highlights this technique with the removal of an instance, which results in the partitions managed by the removed VM being reassigned to the remaining instances. This functionality provides elasticity and to achieve it, the instances need to belong to the same consumer group. Using this observation in SaaMS architecture, it is better understood that a microservice instance uses the same consumer group to take advantage of Kafka Streams reassignment capabilities. Besides that, the functionality of reassignment is used to guarantee fault-tolerance making SaaMS a powerful tool for efficient stream processing in real-time.

4.2. Synopses Library Implementation

The Synopsis Java class is the basic structural component for synthesizing different summaries with high accuracy and performance. It has an abstract construction to handle different implementations for every algorithm using the advantage of polymorphism. Also, it uses the capabilities of inheritance for more flexibility in dynamically adding or removing algorithms without significant changes. The attribute of scalability allows remodeling when it's required. In the next Figure 4.5, the UML class diagram of Synopsis is presented for more details:

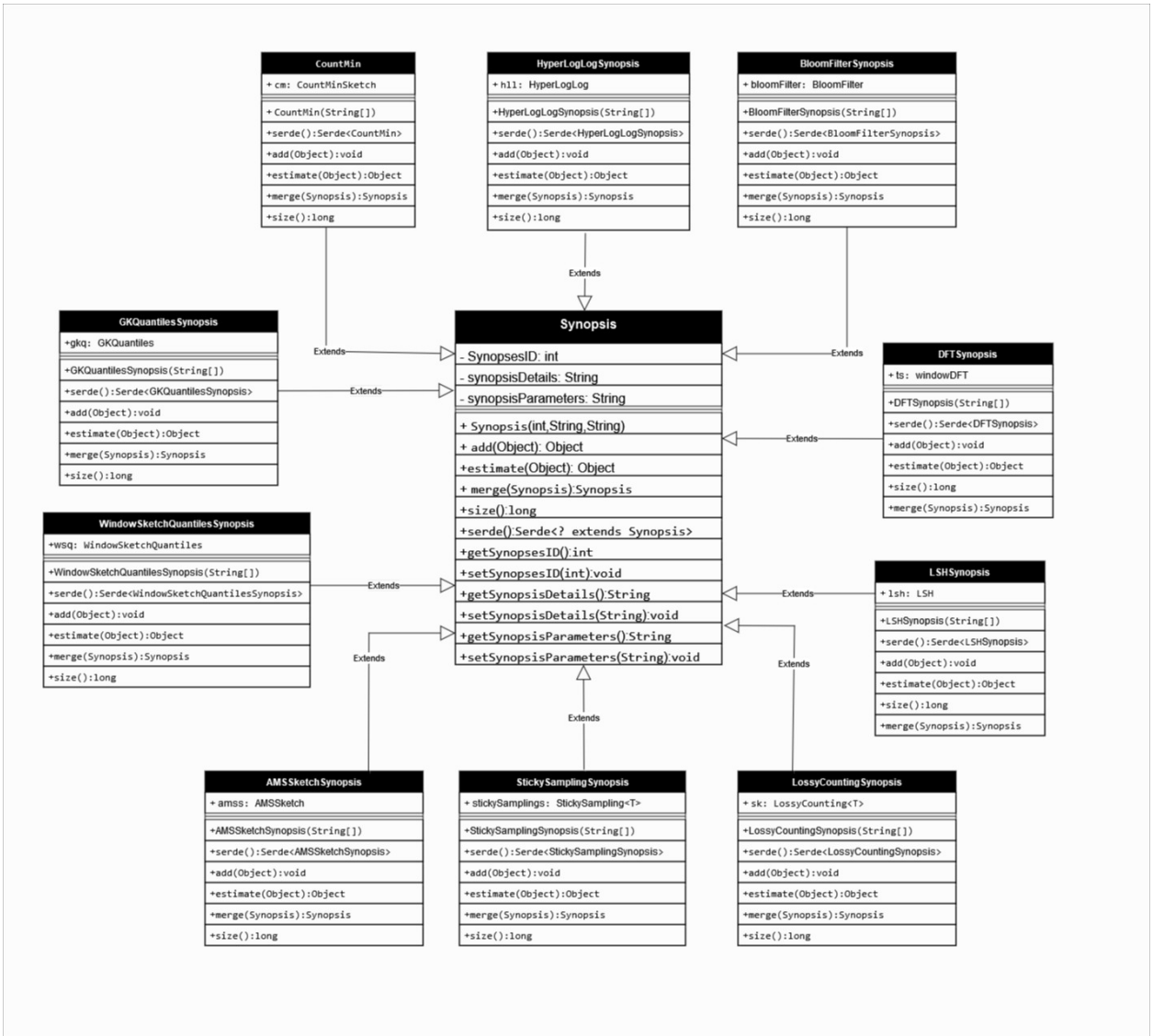


Figure 4.5: SaaMS software architecture

Observing the UML diagram the following analysis appears: There are 3 member variables in the abstract class Synopsis each one used for a specific purpose:

1. The **SynopsisID** is an integer that is used to identify the kind of summarization e.g. 1 for CountMin, 2 for HyperLogLog, etc.

2. The **synopsisDetails** is a string for storing important information about how the synopsis has been built, such as the key of the Create Synopsis Request, the partition it belongs to, and the RequestID.

3. The **synopsisParameters** is a string that contains the parameters previously seen in Section 2.3 which is used to initialize the synopsis.

It also uses a constructor with arguments **SynopsisID**, **synopsisDetails**, and **synopsisParameters** to initialize these member variables for future uses. Finally, it gives the outline of methods that must be implemented by every subclass like add (updating summaries elements), estimate(query a value for a summary), merge(combination of synopses from different sources), size(give the length of a synopsis) and serdes (defined the serialize and deserialize methods of the synopsis).

```
1  public abstract class Synopsis {
2      private int SynopsesID;
3      private String synopsisDetails;
4      private String synopsisParameters;
5      public Synopsis() {
6      }
7      public Synopsis(int SynopsesID,String synopsisDetails,String synopsisParameters) {
8          this.SynopsesID = SynopsesID;
9          this.synopsisDetails=synopsisDetails;
10         this.synopsisParameters=synopsisParameters;
11     }
```

```
12      public abstract Serde<? extends Synopsis> serde();
13      public abstract void add(Object obj);
14      public abstract Object estimate(Object obj);
15      public abstract Synopsis merge(Synopsis synopsis);
16      public abstract long size();
17
18      //getter and setters of SynopsesID,synopsisDetails,synopsisParameters
19       public int getSynopsesID()
20       {return SynopsesID;}
21
22      public void setSynopsesID(int synopsesID)
23      {SynopsesID = synopsesID;}
24
25      public String getSynopsisDetails()
26      {return synopsisDetails;}
27
28      public void setSynopsisDetails(String synopsisDetails)
29      {this.synopsisDetails = synopsisDetails;}
30
31      public String getSynopsisParameters()
32      {return synopsisParameters;}
33
34      public void setSynopsisParameters(String synopsisParameters)
35      {this.synopsisParameters = synopsisParameters;}
36  }
37
```

Also by the UML diagram note that the sub-classes have a field that constitutes each algorithm

implementation e.g. the "cm" variable implements the count Min Sketch Algorithm. In addition, they provide an implementation of the previously mentioned abstract methods in combination with each algorithm instance variable. The next table lists the synopsis techniques supported in the current version of the Synopsis class:

Synopsis ID	Synopsis	Estimate	Mostly Used	Parameters
1	CountMin [15]	Count	Frequent Itemsets	epsilon, confidence, seed
2	HyperLogLog [20]	Cardinality	Cardinality	rsd (relative standard deviation)
3	BloomFilter [7]	Member of a Set	Membership	numberOfElements, maxFalsePositive
4	DFT [40]	Fourier Coefficients	Correlation	intervalSec,basicWindowSize, slidingWindowSize,coefficients
5	LossyCounting [27]	Count, FrequentItems	Frequent Itemsets	epsilon (the maximum error)
6	StickySampling [27]	Count, FrequentItems	Frequent Itemsets	support, epsilon, probabilityOfFailure
7	AMS [1]	L2 norm, Count	Frequent Itemsets	depth, buckets
8	GKQuantiles [21]	Quantile	Quantiles	epsilon (the maximum error)
9	LSH [25]	Binary Representation of a Set	Correlation	slidingwindow(W),compression(D), workersNum(B)
10	WindowSketch Quantiles [5]	Quantile	Quantiles	epsilon (the maximum error),windowSize

Table 4.1: Synopsis Table

4.3. User Interface Implementation

SaaMS application is a user-friendly environment and doesn't require special skills to use. The interaction is carried out obligatorily with Requests which are structured in JSON format using standard fields like these:

- The **streamID** (String) symbolizes the item that needs to build up a Synopsis, in a financial example, it represents the Stocks name
- The **synopsisID** (Integer) is a number that is used as an identifier for a specific Synopsis algorithm and is defined based on Table 4.1
- The **requestID** (Integer) is a unique identifier to distinguish each request
- The **dataSetKey** (String) represents the source from which the StreamID comes, in a financial example it represents the Stock Market name.
- The **param** (`Object[]`) is a table that contains different parameters necessary to build up a Synopsis (CreateSynopsis or query), including the field of values needed to build up the synopsis and the parameters required to use based on Table 4.1
- The **noOfP** (Integer) defines the parallelization level of Topics and Synopsis Microservice

Relying on this information about fields of request, we will present the two types of messages that are used to accomplish the necessary communication. The first type is the create synopsis request, or the equivalent "NotQueryable" request. For example, building a Synopsis based on stock **EYRTRY** of **Forext** market with parallelism 5 and estimating frequencies of price with the Count Min algorithm can be achieved by sending the following message to the Request[19] topic of SaaMS.

[19]Producing request into the Request topic is obligatory and uses another script which is contained in SaaMS code

```
{
    "streamID": "EURTRY",
    "synopsisID": 1,
    "requestID": 1,
    "dataSetKey": "Forex",
    "param":["CountMin","price","NotQueryable",0.001,0.99,12345],
    "noOfP": 5,
    "uid": 1001
}
```

This is how one can easily create a Synopsis and forward data to handle it, but it is also required to be able to ask for specific values estimation. This happens similarly to the previous example but the request in this case has some differences:

```
{
    "streamID" : "EURTRY",
    "synopsisID" : 1,
    "requestID" : 1,
    "dataSetKey" : "Forex",
    "param" : [ 6.05736, "price", "Queryable", "Continues", 0.001, 0.99, 12345 ],
    "noOfP" : 5,
    "uid" : 1001
}
```

In this example, the query request asks for a continuous frequent estimation for the price 6.05736. The declaration of Continuous and Ad-hoc requests is provided in Section 3.2.4 & 3.2.5. The result

of this estimation is written on an output Kafka topic[20] as has already been described. Also, the **streamID** may be empty if we want to establish a Dataset Synopsis based on Section 3.2.3. The representation of the messages that the output topic can contain is presented below:

```
For Stock EURTRY and Dataset Forex
Estimate Count in the price field of value: 6.05959
Count Min Result is: 37
```

As we observe each estimation contained, the streamID, the dataset it belongs to, the summarization implemented, the value requested for estimation, and finally, the result of query estimation.

In addition, as we have already described in Section 3.2.6 the functionality of loading a saved Synopsis from disc. In practice, this can be implemented using the following request:

```
{
  "param" : [ "LOAD_REQUEST", "PathToLoadSynopsis\\stored_CountMin.ser" ]
}
```

As we observed the only necessary value to set up that request is Load Request and the path for recovering a saved Synopsis, SaaMS will take care of the rest.

Finally, the definition of the type of Data that will arrive at the Data topic is required to have a specific JSON format using standard fields like these:

- The **streamID** (String) is the name of the item on which a synopsis can be built, e.g. stock name
- The **objectID** (String) is a unique identifier for each streamID
- The **dataSetKey** (String) is the source that contains streamID, e.g market name

[20]The result can be consumed from the Output Topic using this command: **bin/kafka-console-consumer.sh –bootstrap-server "Brokers Definition" –topic "TopicName" –from-beginning**

- The **date** (String) and the **time** (String) are the timestamp at which the streamID was captured
- The **price** (Double) is the current price at the date and time specified by the previous variables: date and time
- The **volume** (Integer) is the quantity of current trades of this StreamID

The JSON representation of the data tuple[21] is:

```
{
    "streamID":"EURTRY",
    "objectID":"EURTRY_0",
    "dataSetKey":"Forex",
    "date":"01/02/2019",
    "time":"00:00:01",
    "price":6.0654,
    "volume":1
}
```

[21]The format isn't binding because SaaMS contains a script that can transform txt formats to the required JSON type

4.4. Comparison against SDEaaS

This section will present the features of SaaMS compared with other similar applications to better understand its capabilities. This application is the Synopsis Data Engine as a Service (SDEaaS) [24] which has been constructed with the same requirements but implemented in a different framework. It is built on Flink and Kafka opposite to SaaMS which is built on Kafka and Kafka Streams. The following table contrasts the features of SDEaaS and SaaMS:

Feature	Flink SDEaaS	SaaMS
IoT Support	None	Any device that can ran Java
Horizontal scalability: Synopsis Parallelism	A priori max parallelism at the SDEaaS job level.	Fine-tuned parallelism per synopsis.
Horizontal scalability: Synopsis Elasticity	Considerable downtime for scaling in or out. Elasticity at the level of an entire job.	Zero downtime for scaling in or out. Elasticity at the level of synopsis/microservice.
Vertical Scalability	Predefined task slots at the job level. Exploits pseudo-parallelism and hyperthreading.	Fine tuning tasks for new microservices per synopsis. Exploits pseudo-parallelism and hyperthreading.
Federated Scalability	External via Kafka. Communication cost equivalent to SaaMS for equivalent synopses	Native via Kafka. Communication cost equivalent to SDEaaS for equivalent synopses.
Synopsis Library Customization	Develop and maintenance takes time. Adding code of new synopses on-the-fly (at SDEaaS runtime) is cumbersome due to ClassLoader mandatory usage, potential security issues in YARN-like clusters.	Fast to develop and maintain. Adding code of new synopses on-the-fly (at SaaMS runtime) just creates new MicroService. No need for ClassLoader usage.
Deployment Options	Single Mode: From Scratch.	Modes 1: From scratch, Mode 2: Load saved synopses.
Native API Exploitability	Window processing needs manual coding per synopsis. Cannot use windowing operators of the DataStream API.	Kafka Streams API fully exploitable.
Administration	Automatic cluster administration via YARN, MESOS etc	Manual cluster administration
Failure Handling	Exactly once semantics in case of worker failures	Exactly once semantics in case of VM/thread failures
Out-of-order Handling	Natively supported	Natively supported

Table 4.2: SDEaaS VS SaaMS

CHAPTER 5

EXPERIMENTAL EVALUATION

5.1. Infrastructure Setup

To ensure the performance of SaaMS approach experiments were performed to test the 3 levels of scalability (horizontal, vertical, and federated). To complete these metrics, we use two different machines: a GPU Server and a cluster of computers. Each machine has been transformed to execute SaaMS, meaning that they have a Kafka cluster and a Java environment.

The GPU Server has the following technical characteristics that make it suitable for executing real-world scenarios:

1. Software Configuration
    (a) Kafka Version 2.8.1
    (b) Kafka Streams Version 2.8.1
    (c) Zookeeper Version 3.5.9
    (d) Java Version OpenJDK version 1.8
2. Hardware Configuration
    (a) Two Intel Xeon Silver 4310 processors with 12 cores and 24 threads each
    (b) Four 64GB RAMs RDIMM 3200MT/s each
    (c) One ROM 960GB SSD vSAS with read-intensive 12Gbps
    (d) Two graphic cards NVIDIA Ampere A10, PCIe, 150W equipped with 24GB of memory each

The GPU Server provides an excellent test for realistic processing with high-volume data, which is identical for testing horizontal and vertical scalability but it isn't suitable for federated scalability. For this reason, we test the federated scalability in a cluster with 3 VMs, each having the following technical characteristics:

1. Software Configuration
    (a) Kafka Version 2.8.1
    (b) Kafka Streams Version 2.8.1
    (c) Zookeeper Version 3.5.9
    (d) Java Version OpenJDK version 1.8
2. Hardware Configuration
    (a) One CPU with 8 cores
    (b) One 16GB RAM
    (c) One Rom 30GB

This cluster can test all dimensions of scalability but the reason we didn't use it for performance testing for experiments is the limited resources it has. As a result, it couldn't execute realistic scenarios making necessary the use of GPU Server.

5.2. Assessing Scalability

In this section, the performance of SaaMS in different scenarios will be presented to prove how it is achieved in all dimensions of scalability. To examine horizontal and vertical scalability, we will utilize the GPU Server with a Dataset which contains stocks from different stock markets. The values of these stocks have the format we have seen in Section 4.3. However, the federated scalability will be examined with the use of a Cluster. To begin with, examination of horizontal and

vertical scalability is necessary to calculate Throughput[22] while we change either parallelism degree or the number of streams. On the other hand, the federated scalability is computed by measuring Communication Cost[23] while we alter the number of VMs.

In each figure, a type of synopses (CountMin, HyperLogLog, and Discrete Fourier Transform) will be built and maintained. These summarization techniques have already been presented in Section 2.3. For better understanding, the synopses parameters are represented both practically and theoretically to avoid confusion.

1. The **CountMin in SaaMS** uses (epsOfTotalCount,confidence) = $(0.002, 0.99)$, where these parameters are connected with the theoretical analysis transformed $(\epsilon, \delta) = (0.002, 0, 01))$ [24]

2. The **HyperLogLog in SaaMS** in practice uses one parameter: (rsd) = (0.02). Additionally, the previous theoretical analysis of HyperLogLog utilizes one parameter [25]$b = 11$

3. The **DFT in SaaMS**, uses the following parameters:
   (bw,sw,coe) = $(b, w, n) = (500, 200, 8)$

[22]The throughput defined as the number of data tuples processed per second

[23]As communication cost is determined as the measure of the average number of messages exchanged across VM for performing calculations

[24]The difference about confidence andδ is due to how different definitions of the width are in theoretical and practical implementation

[25]The number of b resulting from $b = \log_2 \left(\left(\frac{1.04}{\text{RSD}} \right)^2 \right)$

5.2.1. Throughput versus Number of Workers

In Figure 5.1 is observed that the more we increase the parallelism the more throughput numbers grow. For this experiment, we used a static number of streams equal to 100. The results are as expected based on what we have already discussed about horizontal scalability. In the schema, there is an increase of workers from 2 to 18, resulting in the rise of workers escalating the tuples per second where they are processed.

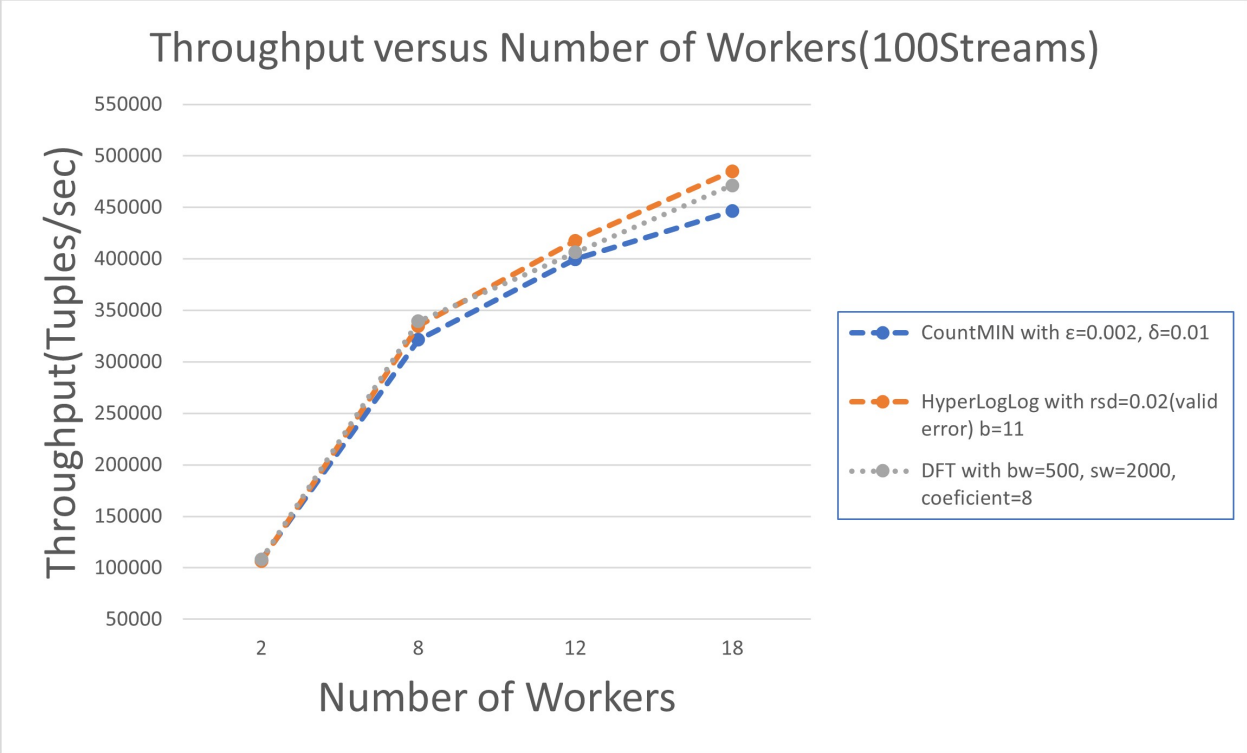


Figure 5.1: Throughput versus Number of Workers

5.2.2. Throughput versus Number of Streams

In Figure 5.2, we also examined the rise in throughput upon increasing the number of processed streams from 30 to 150. In this diagram, we used a static number of workers equal to 9. These outcomes validate our claim about vertical scalability and how SaaMS takes advantage of Synopsis to maintain them without performance bottlenecks.

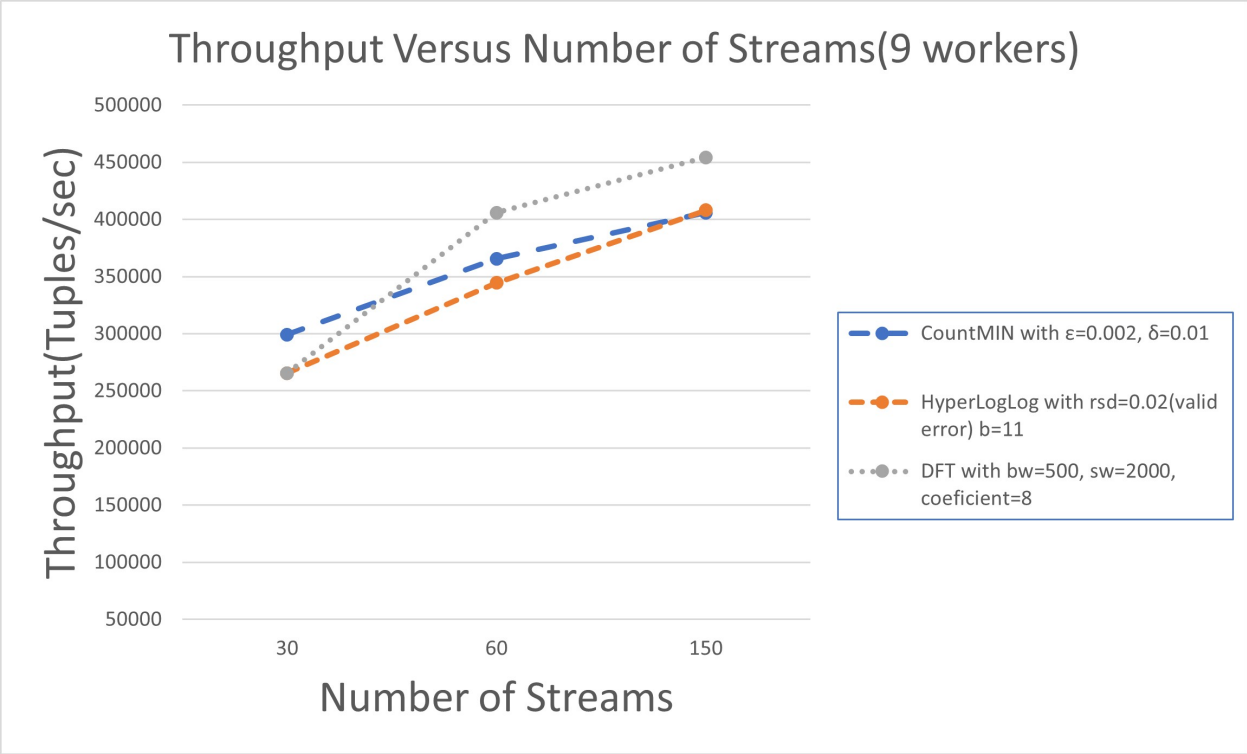


Figure 5.2: Throughput versus Number of Stream

5.2.3. Communication Cost versus Number of VMs

In this section, the cost of communication in bytes (normalized by a logarithm) across the VMs is represented to estimate the final merged synopsis. In Sections 4.1.4 and 4.1.5, we have already presented the implementation of SaaMS in a distributed environment to succeed in federated scalability. It was clear from this analysis that each VM will manage a sub-synopsis and as a final step, they will be merged to generate the final result. The cost of merging these Synopses will be represented in Figure 5.4 using the cluster with 3 VMs for the experiments. More specifically, we are calculating the total size of bytes needed to be transferred from VMs to produce the final summary for this type of Synopsis (CountMin, HyperLogLog, and DFT). The choice of VM where will be implemented the merging process becomes random and also, the VMs don't communicate all the time, but periodically with the use of a punctuator every 20 seconds. Furthermore, to better understand the efficiency of utilized synopses, we have demonstrated in the same figure the communication cost if we are using raw data. The vertical axis is in logarithmic scale to stabilize variance. As is obvious,

the use of Synopsis provides great compression and decreases the cost of VMs communication.

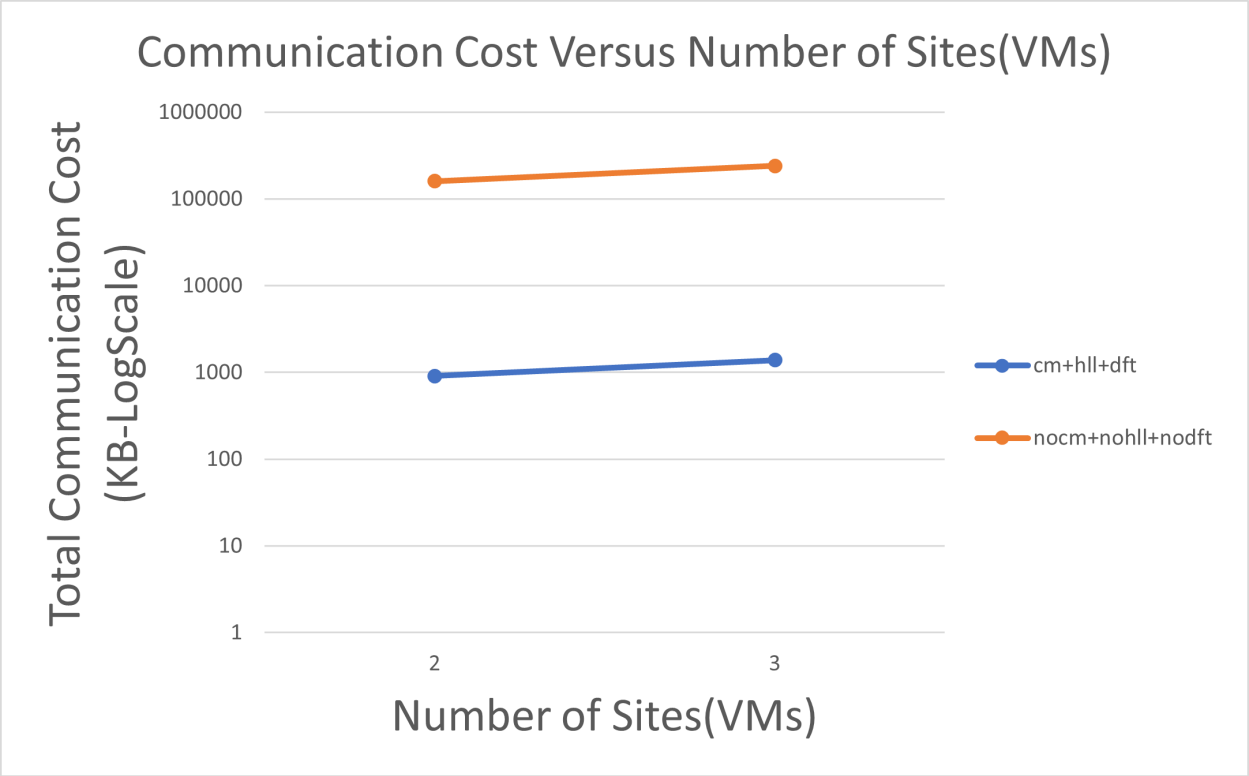


Figure 5.3: Communication Cost versus Number of Sites(VMs) either in case of use of Synopsis and either in case of use of Raw data

Moreover, apart from the experiments presented previously and performed in the cluster, we will be implementing the same experiments on the GPU Server. As we already know, the GPU Server doesn't have VMs as a result in practice the term "Communication Cost" doesn't exist but we will adopt it abusively to present a real-world scenario. In Figure 5.4 communication cost is presented versus "VMs" (in fact it is not VMs but threads):

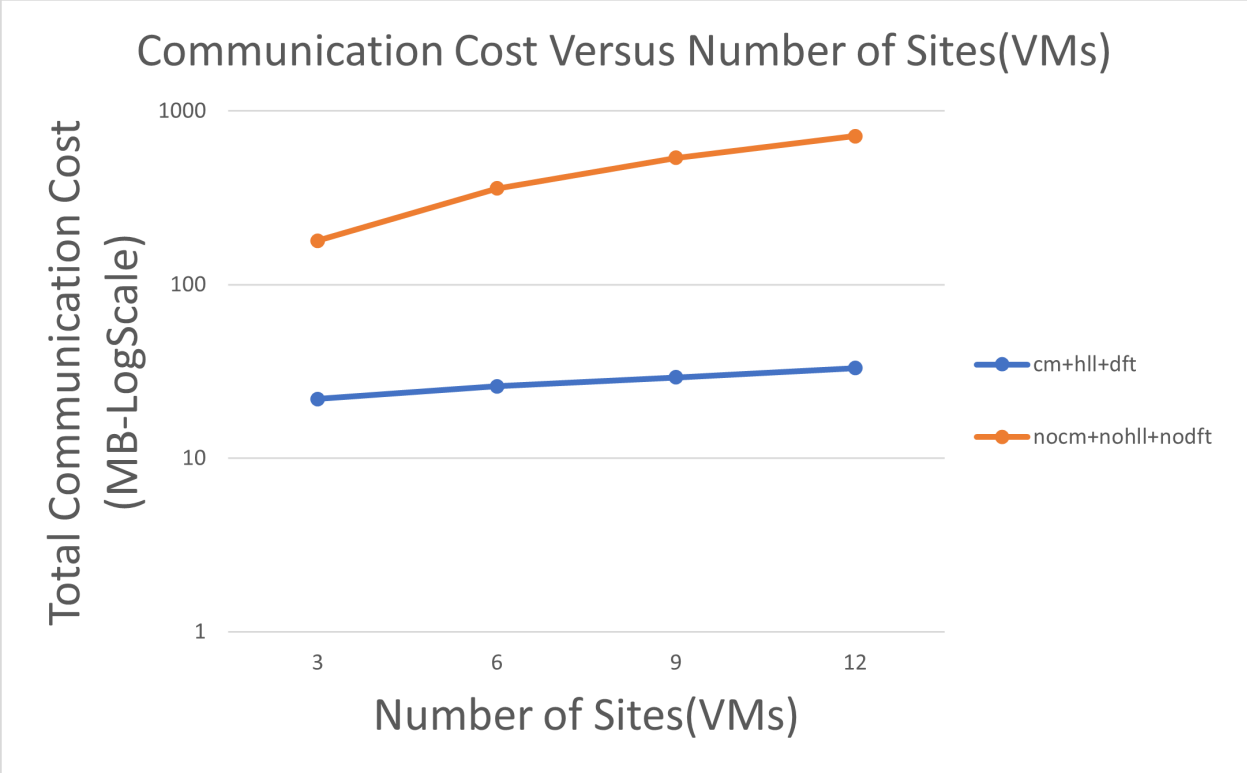


Figure 5.4: Communication Cost versus Number of Sites(VMs) in GPU Server

As before the use of Synopsis compresses the data and decreases the cost of communication across the "VMs".

CHAPTER 6

CONCLUSION & FUTURE WORK

In conclusion, Synopses as a MicroService (SaaMS) project is a powerful tool that uses the capabilities of Kafka and Kafka Streams for achieving real-time data processing. The innovation in this application is that we have built a novel computational paradigm that provides all types of required scalability adding up elastic resource allocation with zero downtime and the ability to get deployed on any Java-enabled device. The ability to implement these processes in real-time with high throughput and low latency makes it effective in real-life examples.

The upgrades will be developed in the future for SaaMS are directed towards the following:

- ✔ The first update in SaaMS must be the extension of the summarization that the Synopsis class includes in the app. As Table 4.1 shows, currently 10 types of synopsis are implemented which are used to find frequent items, cardinality, correlation, and quantiles. Except for frequency item estimation the other types have only one algorithm so would be helpful to add more algorithms in each category to suit our use case. Also, it would be beneficial to enrich the Synopsis class with new categories like stream join Synopses, graph-based Synopses, time windowed Synopses, etc.

- ✔ The second upgrade would be improvements to the user interface. In the existing project, there are two functionalities: build synopses and estimate queries based on specific synopses. It would be useful to fit into the project more functionalities like listing the total active synopses, deleting one synopsis, and changing the current API to make these processes more automated.

- ✔ The third improvement could be upgrading the way it communicates with users. Instead of using a console, a GUI environment could be built to make the interaction less complicated. This would make the application more accessible to more people.

Bibliography

[1] Noga Alon, Yossi Matias, and Mario Szegedy. "The Space Complexity of Approximating the Frequency Moments." In: *Journal of Computer and System Sciences* 58 (1999), pp. 137–147.

[2] Arnd Christian König Anshumali Shrivastava and Mikhail Bilenko. "Time adaptive sketches (Ada-Sketches) for summarizing data streams." In: (2016), pp. 1417–1432.

[3] *Apache Kafka*. URL: https://kafka.apache.org/.

[4] *Apache Kafka Streams*. URL: https://kafka.apache.org/documentation/streams/.

[5] Arvind Arasu and Gurmeet Singh Manku. "Approximate counts and quantiles over sliding windows." In: *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 2004.

[6] Brian Babcock, Mayur Datar, and Rajeev Motwani. "Sampling from a moving window over streaming data." In: (2002).

[7] Burton H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors." In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[8] Confluent. *Crossing Streams: Joins in Apache Kafka*. Confluent Blog. 2023. URL: https://www.confluent.io/blog/crossing-streams-joins-apache-kafka/.

[9] Confluent. *Getting Started with Kafka Streams*. Confluent Blog. 2023. URL: https://developer.confluent.io/courses/kafka-streams/get-started/.

[10] Confluent. *Introducing Kafka Streams: Stream Processing Made Simple*. Confluent Blog. 2023. URL: https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/.

[11] Confluent. *Kafka Streams Architecture - Parallelism Model*. Confluent Blog. 2023. URL: https://docs.confluent.io/platform/current/streams/architecture.html#streams-architecture-parallelism-model.

[12] Confluent. *Kafka Streams Architecture: Parallelism Model*. Confluent Blog. 2023. URL: https://docs.confluent.io/platform/current/streams/architecture.html#streams-architecture-parallelism-model.

[13] Confluent. *What is an Apache Kafka Cluster?* Confluent Blog. 2023. URL: https://www.confluent.io/blog/what-is-an-apache-kafka-cluster/.

[14] Confluent. *What is Apache Kafka?* Confluent Blog. 2023. URL: https://developer.confluent.io/what-is-apache-kafka/.

[15] Graham Cormode and S. Muthukrishnan. "An Improved Data Stream Summary: The Count-Min Sketch and its Applications." In: *Journal of Algorithms* 55 (2005), pp. 58–75.

[16] Hevo Data. *Everything You Need to Know About Kafka Clusters*. Hevo Blog. 2023. URL: https://hevodata.com/learn/kafka-clusters/#t6.

[17] Giselle van Dongen and Dirk Van den Poel. "Evaluation of Stream Processing Frameworks." In: *IEEE Transactions on Parallel and Distributed Systems* 31.8 (2020).

[18] University of Edinburgh. *Lecture Notes on Data Streams*. 2023. URL: https://www.inf.ed.ac.uk/teaching/courses/exc/slides/dataStreams02.pdf.

[19] Facebook Engineering. *HyperLogLog: Data Infrastructure at Facebook*. 2018. URL: https://engineering.fb.com/2018/12/13/data-infrastructure/hyperloglog/.

[20] Philippe Flajolet et al. "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm." In: *Proceedings of the 2007 Conference on Analysis of Algorithms*. DMTCS. 2007, pp. 127–146.

[21] Michael Greenwald and Sanjeev Khanna. "Space-Efficient Online Computation of Quantile Summaries." In: *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. ACM. Santa Barbara, California, USA, 2001, pp. 58–66.

[22] *How to Encourage Farmers to Use Big Data Analytics in Agriculture*. Intellias Blog. 2023. URL: https://intellias.com/how-to-encourage-farmers-to-use-big-data-analytics-in-agriculture/.

[23] Taiwo Kolajo, Olawande Daramola, and Ayodele Adebiyi. "Big data stream analysis: a systematic literature review." In: *Journal of Big Data* 6.47 (2019).

[24] Antonios Kontaxakis. "DESIGN AND IMPLEMENTATION OF A DISTRIBUTED SYNOPSIS DATA ENGINE ON APACHE FLINK." Master's thesis. Technical University of Crete, 2020.

[25] Antonios Kontaxakis et al. *windowDFT.java in SDE repository*. GitHub repository source code. 2020. URL: https://github.com/akontaxakis/SDE/blob/master/src/main/java/infore/SDE/synopses/Sketches/LSH.java.

[26] Maycon Bordin & André Laszlo. *Streaminer: A collection of algorithms for mining data streams*. GitHub repository. 2020. URL: https://github.com/mayconbordin/streaminer.

[27] Gurmeet Singh Manku and Rajeev Motwani. "Approximate Frequency Counts over Data Streams." In: *Proceedings of the 28th VLDB Conference*. Hong Kong, China, 2002, pp. 346–357.

[28] Vitter Jeffrey Scott Matias Yossi and Wang Min. "Wavelet-based histograms for selectivity estimation." In: (1998), pp. 448–459.

[29] Micvog. *Frequency Counting Algorithms Over Data Streams*. 2015. URL: https://micvog.com/2015/07/18/frequency-counting-algorithms-over-data-streams/.

[30] QuantHub. *What Are the Two Main Methods to Summarize Data?* 2023. URL: https://www.quanthub.com/what-are-the-two-main-methods-to-summarize-data/.

[31] *Real-Time Data in Health Informatics*. Northern Kentucky University, MSHI Program. 2023. URL: https://onlinedegrees.nku.edu/programs/business/informatics/mshi/real-time-data/.

[32] TIBCO Spotfire. *What is Data Streaming?* https://www.spotfire.com/glossary/what-is-data-streaming. 2023.

[33] *Stock Market Data Analytics*. Vention Teams Blog. 2023. URL: https://ventionteams.com/blog/stock-market-data-analytics.

[34] *Stream Processing Frameworks Compared: Top Tools for Processing Data Streams*. Nexocode Blog. 2023. URL: https://nexocode.com/blog/posts/stream-processing-frameworks-compared-top-tools-for-processing-data-streams/.

[35] *Stream-lib: Summarizing data in stream*. GitHub repository. 2010. URL: https://github.com/addthis/stream-lib.

[36] Vladimir Topolev. *Kafka Overview with pictures*. Geek Culture. 2021. URL: https://medium.com/geekculture/essential-kafka-overview-with-pictures-bffd84c7f6ac.

[37] Lu Wang et al. *Quantiles over Data Streams: An Experimental Study*. URL: https://speakerdeck.com/coolwanglu/quantiles-over-data-streams-an-experimental-study.

[38] Wikipedia. *Bloom filter*. Wikipedia, the free encyclopedia. 2023. URL: https://en.wikipedia.org/wiki/Bloom_filter.

[39] Wikipedia contributors. *Data stream management system*. 2023. URL: https://en.wikipedia.org/wiki/Data_stream_management_system.

[40] Yunyue Zhu and Dennis Shasha. "StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time." In: *Courant Institute of Mathematical Sciences, Department of Computer Science, New York University* (2002).