DELFT UNIVERSITY OF TECHNOLOGY

MASTERS THESIS

Estimation of Similarity Between Data Streams Using Probabilistic Data Structures

Author:
Panagiotis REPPAS

Supervisor:
Dr. Asterios KATSIFODIMOS

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Web Information Systems Group
Software Technology

April 7, 2024

Declaration of Authorship

I, Panagiotis REPPAS, declare that this thesis titled, "Estimation of Similarity Between Data Streams Using Probabilistic Data Structures" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science
Software Technology

Master of Science

Estimation of Similarity Between Data Streams Using Probabilistic Data Structures

by Panagiotis REPPAS

This thesis embarks on the quest to efficiently compute similarities between data streams in real-time, a task burgeoning in importance with the advent of big data and real-time analytics. At the heart of this endeavor is the expansion of the Condor framework to accommodate new probabilistic data structures, tailored to meet the distinctive challenges posed by streaming data. A notable highlight is the adaptation of the DSTree data structure to a streaming environment, marking a significant stride towards achieving the stated goal. Through an implementation within the Condor framework, this research explores the core mechanisms for indexing and approximating similarities, paving the way for more refined analyses. Furthermore, a comparative study is conducted encompassing several probabilistic data structures, including HyperLogLog and Theta Sketches, examining their effectiveness in similarity search within a streaming environment, in comparison with the DSTree method. The evaluation of these methods will be done through a series of experiments, which are meticulously designed to measure the accuracy and efficiency of these structures, shedding light on their potential and limitations. he insights garnered from this study underscore the potential of probabilistic data structures in bolstering the speed and accuracy of similarity search in streaming data, while also hinting at promising avenues for further research.

Acknowledgements

First and foremost, I wish to express my deepest gratitude to Prof. Katsifodimos, whose guidance, support, and expertise were instrumental in the shaping and completion of this research.

I would also like to extend my appreciation to George Siachamis, who as an advisor provided not only critical feedback but also consistent encouragement, aiding me at every step of this research process.

My sincere thanks go to TU Delft for providing the environment and resources that have been crucial in bringing this thesis to fruition. Being part of this esteemed institution has been a significant chapter in my academic journey.

On a personal note, I would like to acknowledge my parents, whose faith in my capabilities and continuous support has been the foundation on which all my endeavors are built. I also want to express my gratitude to my friends, whose camaraderie and occasional reality checks have kept me grounded throughout this experience.

Contents

List of Figures

Chapter 1

Introduction

In recent years, our digitized world has witnessed an unprecedented surge in the volume of produced data. From the non-stop stream of social media updates to the relentless flow of sensor-generated data, the digital footprints of human and machine interactions is expanding exponentially, creating a vast and ever-growing landscape of information. The speed with which the data is being generated has also dramatically increased, mostly because it is produced by non-human entities, such as sensors or bots, which can produce data at much higher pace than humans. Finally, the digital era has given rise to a plethora of connected devices, online platforms, and sensor networks, each contributing to the abundance of real-time information flowing through the digital landscape [8].

This threefold increase – in volume, velocity, and variety – while it offers immense potential for insights, innovation, and informed decision-making, it also introduces significant challenges. The sheer scale and complexity of managing, processing, and analyzing this amount of information necessitate the development of advanced and efficient computational methodologies, especially in the realm of data stream analysis. Data streams, characterized by their continuous and unbounded nature , are a cornerstone of this data-rich environment. This is due to the fact that in more and more applications are time-sensitive and processing speed impacts greatly the quality of service. For example, seamless and efficient traffic among connected vehicles or real-time fraud detection, highlight the increasing importance of data stream analysis, as they depend on the transfer and processing of a vast amount of data in real-time.

To address these challenges, researchers have turned towards probabilistic data structures as a promising solution to explore. These structures find a balance between computational efficiency and a certain level of approximation, making them adept at handling the processing of large data streams with fewer resource requirements compared to traditional methods. This attribute makes them especially fitting for the domain of streaming data analysis.

This thesis investigates the application of probabilistic data structures in the realm of data stream analysis, with the goal of devising methods for effectively measuring similarity between various data streams.

1.1 Problem Statement

Data streams are sequences of data generated in a continuous manner. Determining the similarity between data streams is crucial for a variety of practical reasons across multiple sectors. Areas such as finance and healthcare require immediate decisions and responses, thus making real-time analytics a necessity. Similarity calculation is also used in applications where the goal is anomaly detection, playing a vital role in identifying issues like network breaches and fraudulent financial transactions. For

example, streams from network traffic can be used to estimate similarity between network packets to identify malicious ones, or streams from financial transactions can be used to spot unusual behavior and thus fraudulent transactions. Moreover, stream similarity estimation can improve the quality of personalized recommendations in online services, enhancing overall user satisfaction. Streams from user activity can be used to estimate similarity between users and provide more efficient recommendations on products and services. To sum up, assessing the similarity between data streams is a key task that yields valuable insights and informed decision-making across diverse applications and industries. Consequently in order to make data analysis algorithms and pipelines more scalable, it is important to develop more efficient similarity search techniques.

1.2 Possible Solutions

Traditionally, data analysis techniques require the data to be stored and then analyzed off-line using algorithms that make several passes over the data and are characterized by high complexity. However, data streams are by nature infinite, and data are generated with high rates, making it impossible to store in main memory. It is difficult to process the high velocity, high volume, and high variety data with SQL-based tools and as a result, alternative techniques have to be considered in order to solve the aforementioned issues.

One solution is the use of probabilistic data structures. Probabilistic data structures are a class of data structures that, as their name suggest uses randomness in order to achieve certain properties. While, they do not guarantee to give correct answers every single time, they can provide approximate answers with a high degree of accuracy, while using less memory and time. This makes them ideal for applications where it is more important to get an answer quickly than to get a perfectly accurate answer.

The second solution that we are going to investigate is by adopting techniques used in time series data analysis. Time series data is a collection of quantities that are assembled over even intervals in time and ordered chronologically. This type of data can be found in nearly every domain such us engineering, finance, medicine etc [21] [29]. As it can be easily understood, data streams and time series data are two closely related concepts and share a lot of similarities. It was only natural that we considered time-series analysis techniques as a potential candidate to tackle the problems that arise in data streams analysis.

For the purposes of this project, the aforementioned solutions will be investigated. The first is to use probabilistic data structures in order to create a summary of a data stream for separate time intervals. Then, use these summaries to calculate the similarity between data streams. The second approach, is to use an algorithm designed for calculating similarity in time series data and modify it in a way that allows streaming data analysis. The idea behind this approach is that data that arrive from each separate data stream can be processed in a way similar to time series data.

1.3 Research Questions

Based on the problem statement a number of research questions can be derived. The central inquiries driving this investigation are articulated below, organized to mirror the procedural flow of the research:

- Among the array of probabilistic data structures, how do particular structures like HyperLogLog and Theta Sketches perform in computing similarities between data streams?
- How can a state of the art time series processing method be altered to efficiently handle streaming data while maintaining its core functionality for similarity search?
- How do the implemented data structures and algorithms stack up against each other in terms of efficiency, accuracy?
- What insights emerge from the experimental assessments regarding the suitability of these data structures for similarity search in streaming data?

1.4 Contributions

In this section, we outline the contributions that emerged in the process of answering our research questions in the same order, as expressed in the previous section.

A notable endeavor within this work is the expansion of the Condor framework to seamlessly incorporate a streaming version of DSTree and Theta Sketches. These addressed challenges related to harmonizing these data structures with streaming data, paving the way for a more robust similarity search mechanism within the framework. A central contribution is the modification and implementation of a streaming version of the DSTree. This new version of DSTree is crafted to manage streaming data, with a special focus on efficiently determining distances between different data streams.

Furthermore, this thesis embarked on a comprehensive evaluation to compare several data structures through various experiments. The goal was to ascertain their appropriateness for similarity search in a streaming environment. The analysis offers insights into the performance and efficacy of these data structures, enriching the understanding of their potential and limitations in real-time data stream analysis.

Through these contributions, this thesis has laid a solid foundation for further exploration and refinement of probabilistic data structures in streaming data analysis within the Condor framework.

1.5 Outline

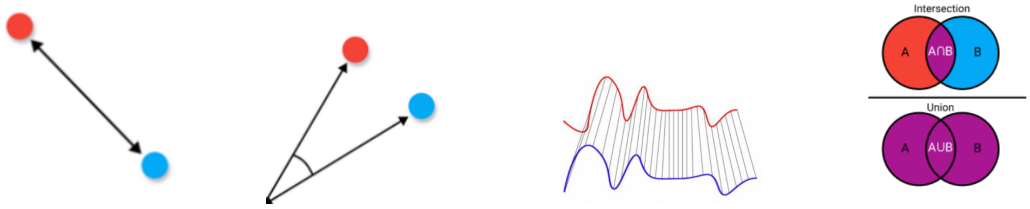
The subsequent sections of the thesis are organized as follows. Chapter 2 undertakes a literature review on probabilistic data structures viable for calculating similarity between varying data streams, alongside time series analysis techniques that hold the potential to be adapted for use in a streaming data environment. Chapter 3 contains the design considerations and the development of an API which enables users to select from a few renowned probabilistic data structures, and a new one devised specifically for this work. It's clear that the inspiration for the new data structure stemmed from a time series analysis technique, which was then tailored to fit a streaming scenario. Following that, Chapter 4 presents the experiments carried out to evaluate these methods. The experiments were designed in order to measure both the efficiency and the accuracy of each approach. In the same chapter, the results of theses experiments can also be found. Lastly, Chapter 5 provides a conclusion to the thesis, encapsulating our work, acknowledging its limitations, and suggesting avenues for subsequent research.

Chapter 2

Literature Review

Over recent decades, data stream analysis and processing have evolved into a major area of focus across such as financial data analysis [36], sensor network monitoring [37], and network traffic analysis [24], among others. Many applications require the on-line detection of hidden patterns that may exist in different data streams, thereby underscoring the need for algorithms that will calculate the similarity between these streams accurately and in timely fashion. To achieve this, there are typically usually two requirements: a measure to quantify similarity and scalable algorithms to estimate these measures. In a data stream environment scalability can often refer to parallelization of the algorithms in order to ensure accelerated execution times.

Regarding the first requirement of estimating similarity between data streams, several metrics have been examined in order to calculate the similarity between two data streams by measuring the distance between them in a feature space. Some common distance-based measures include the Euclidean distance, cosine distance, dynamic time warping (DTW), and Jaccard distance. The Euclidean distance [38] is the classic point-to-point distance between two points in a feature space. It is calculated by taking the square root of the sum of the squared differences between the two points in each dimension. Dynamic Time Warping (DTW) [33] is particularly useful for comparing time series data, where the feature space is a sequence of values over time. It is calculated by finding the alignment between the two sequences that minimizes the distance between them. It possesses an interesting feature, as it can account for shifts in time between the two data streams, albeit at the cost of added complexity and thus a slowdown in response time. The cosine distance [26] is often used when dealing with textual data, where the feature space is a vector of word counts. It is calculated by taking the cosine of the angle between the two vectors. Finally, Jaccard Distance [5] is calculated by the ratio of the intersection size to the union size of two sets, providing a measure of similarity between the sets.



(A) Euclidean (B) Cosine (C) DTW (D) Jaccardian

FIGURE 2.1: Common distance metrics

Each of these distance measures serves to quantify similarity in different types of data, and their choice would be contingent on the nature of the data streams and the specific requirements of the analysis at hand.

When it comes to the second requirement and the algorithms that can be used in a data stream context, it is not uncommon to handle data streams as time series and employing similar analytical techniques. Yet, the inherent complexity and distinct properties of data streams pose challenges that prevent a direct application of these algorithms to process the streaming data. The evolution into a distributed version facilitates the parallel processing of data streams, harnessing the power of multiple computing resources working simultaneously [38] [15].

Moreover, as mentioned earlier, when it comes to data stream processing, probabilistic data structures emerge as an extremely useful tool, as they can handle large amounts of data in real-time and provide approximate answers to queries with minimal space usage and computation requirements. A key driver behind this capability is their inherent design that often encompasses independent operations, which can be executed simultaneously on distinct segments of the data structure, thereby unlocking substantial parallelization opportunities. Another advantage of probabilistic data structures for parallelization is that they are often approximate. This means that they can be used to obtain approximate answers to queries quickly, without having to process all of the data in a sequential order. Their structure allows different threads to work on different parts of the data simultaneously and then combine their results to produce an approximate answer, which translates to significant performance improvements. This feature will be covered in greater detail in Chapter 3, explaining how this feature helps in handling streaming data faster while keeping a good balance between being accurate and not using too much computing resources. Consequentially, researchers have turned to various probabilistic data structures to tackle to address a range of issues in data stream environments. For instance, they've used these structures to count the unique values in the incoming data [18], estimate how often certain elements appear in a data stream [7], approximate the membership of a set [10], etc. These data structures have shown promise in providing efficient solutions while managing the constant flow of data, showcasing their potential in dealing with the dynamic nature of streaming data.

2.1 Probabilistic Data Structures

In the domain of data analysis, probabilistic data structures offer efficient computational solutions for a variety of challenges. The next step is identifying structures from this diverse pool that are adept at calculating similarities between data sets. The selection process is guided by two primary considerations. Firstly, the size and complexity of the opperations since the streaming environment in which they will take place, dictates that results have to be produced in real-time. Different structures possess varying capacities and efficiencies, with some being inherently more suitable for handling extensive or complex data sets than others. The second factor relates to the specific similarity metric that ill be used. Although numerous metrics are available, the focus of this research is on the Jaccard similarity. So, the challenge goes beyond just picking a structure that can handle the data's volume and complexity. It's also about finding one that's tailored to accurately calculate Jaccard similarity. It also involves selecting the ones that are specifically tailored to facilitate accurate Jaccard similarity calculations. Balancing these criteria is essential to ensure both the reliability and validity of computational results in the context of data stream similarity assessment.

Two prominent probabilistic data structures that have caught the attention for their efficacy in addressing these challenges are HyperLogLog (HLL) and Theta

Sketches. HyperLogLog is renowned for its ability to estimate cardinalities with a relatively small memory footprint, making it a preferred choice for handling large-scale data streams. On the other hand, Theta Sketches offer a flexible framework, facilitating various set operations, and are particularly adept at approximating set sizes and Jaccard similarities. Both structures, with their unique features and computational strengths, will be explored and analyzed to determine their suitability and performance in calculating similarities between data streams. The examination of their underlying algorithms, accuracy, and resource efficiencies will provide insights into their applicability in real-world scenarios.

2.1.1 HyperLogLog

HyperLogLog (HLL) is a probabilistic data structure that stands out for its ability to estimate cardinalities, or the number of distinct elements in a dataset, with impressive accuracy while consuming only a small amount of memory. This unique feature makes it particularly valuable for large-scale data stream processing where traditional counting methods may be infeasible due to memory constraints. Originally proposed by Morris [27] and analyzed by Flajolet [14], HyperLogLog employs hash functions to achieve its estimations. When a new element enters the dataset, it is hashed to produce a binary string. The number of zeros at the beginning of the string is of particular interest because the probability that a given hashed value ends in at least i zeros is $1/2^i$. Intuitively, the farther to the left the first 1 appears, the more likely the item is a rare or unique element. By keeping track of the maximum position of the number of zeros across all elements in the dataset, HLL can infer the number of unique elements.

However, relying on a single maximum position could introduce variability, only one outlier entry whose hash value has too many consecutive zeros will produce a drastically inaccurate (overestimated) estimation of cardinality. Additionally, this can give only a power of two estimate for the cardinality and nothing in between. To counteract this, HLL divides the input data into multiple subsets or "buckets." Each bucket maintains its number of zeros, and the overall cardinality is estimated by averaging across these buckets. This division and averaging process not only improves accuracy but also smoothens out potential outliers. As suggested by Frajolet et al. the data can be assigned to buckets by using the first few (k) bits of the hash value as an index into a bucket and compute the longest sequence of consecutive zeros on the remaining bits.

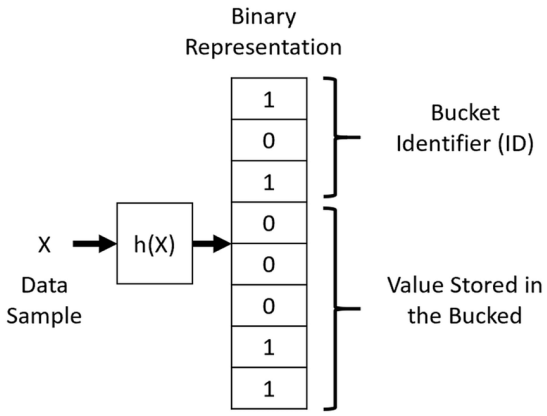


FIGURE 2.2: Depiction of HyperLogLog flow (reproduced from [25]).

In the same paper, it was also observed that outliers greatly decrease the accuracy of this estimator. Thus, the accuracy can be improved by throwing out the largest values before averaging. Finally, using harmonic mean instead of the geometric mean, edges down the error rate even further with no increase in required storage. The error rate is slightly less than, $1.04/\sqrt{m}$. Where m is the number of registers.

One of the standout features of HLL is its memory efficiency. The number of buckets can be adjusted based on the available memory, and each bucket typically requires only a few bits of storage to track the number of leading zeros. HLL sketches can calculate cardinality using only $O(log_2 log_2(n))$ space, this is also from where the data structure took its name. Additionally, HLL supports the merging of multiple structures. If two HLL structures have processed different parts of a dataset, they can be seamlessly merged to produce a cardinality estimate for the combined dataset. This property is especially advantageous in distributed systems, where data might be processed across different nodes.

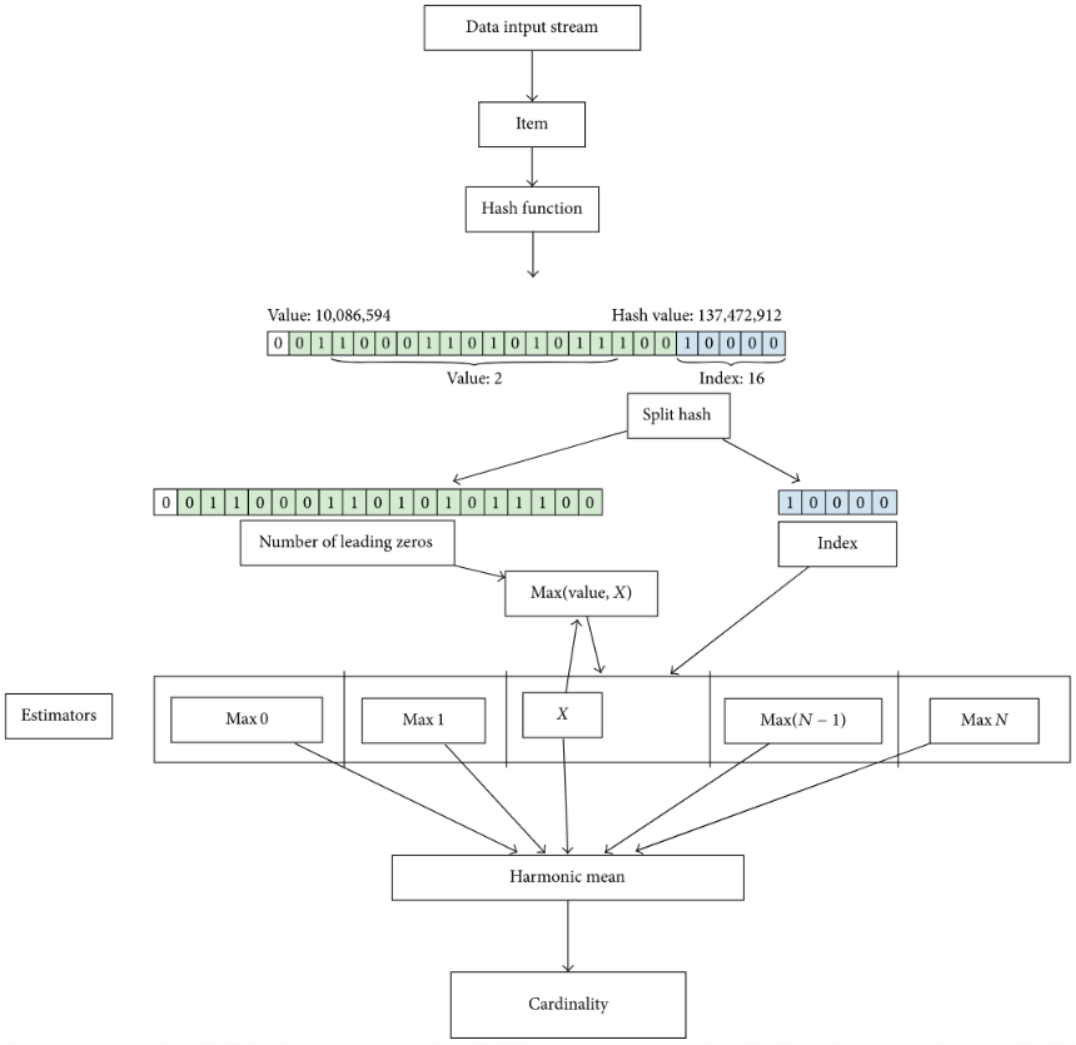


FIGURE 2.3: Data-flow depicting all the steps of the HyperLogLog algorithm. Source: [17]

However, like all probabilistic data structures, HLL has a trade-off between accuracy and memory usage. While it provides a high degree of precision with minimal memory, there's still an error margin. The error rates, usually a few percent, can be reduced by allocating more memory to the structure, but they can never be entirely

eliminated. Moreover, HLL was designed as a means to estimate the cardinality of a data stream and as such it cannot be used in order to estimate similarity. However, there is a work around this handicap. Using some of its properties in order to calculate the union of the produced sketches and the inclusion-exclusion principle, as it will be studied in more depth in Chapter 3.3.1. These extra steps introduce more complexity and increase the error rate of the solution.

In summary, HyperLogLog is a data structure that offers a compelling solution to the cardinality estimation problem in large-scale data-sets. The are two features that render HLL a viable solution in the studied problem. Firstly, its ability to handle huge amounts of data and create synopsis that reduces the size of the original data by several orders of magnitude. Secondly, the fact that the process of creating new synopses can be broken down to smaller pieces that are executed independently, is a necessary feature when dealing with streaming data as most of the operations are executed in a distributed way.

2.1.2 Theta Sketch

The second data structure that will be investigated has completely different properties. In contrast with HyperLogLog, Theta Sketches [9] represent a flexible and efficient probabilistic data structure devised for the purpose of approximating set operations, specifically cardinalities of unions, intersections, and differences. Their utility shines when managing vast data streams, where exact computations may be resource-intensive or simply unfeasible. While, technically Theta Sketches do not define a single data structure, but a framework where its instantiations must specify a threshold choosing function, a combining function and an estimator function. More specifically, as the name suggests, the threshold function is a process that prunes elements that do not hold a specific property, the combining function takes as input a collection of Theta Sketches and returns a single Theta Sketch that is the union of these sketches, and finally, A estimator function that takes as input a Theta Sketch and returns an estimate of the unique hashed stream items present.

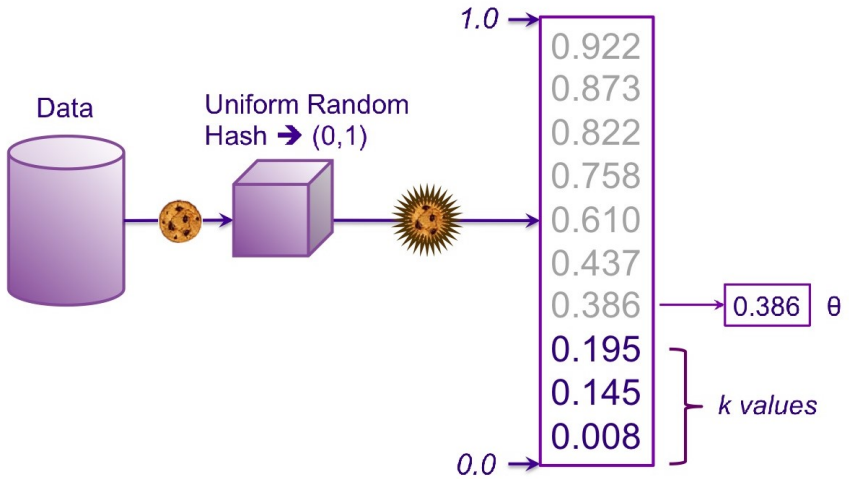


FIGURE 2.4: Representation of the Theta Sketch data structure. Adapted from [2]

In our case, Theta Sketches will be a generalization of the Kth Minimum Value (KMV) sketch [4]. The core idea behind Theta Sketches is rooted in the concept of randomized algorithms. Each incoming item in a data stream is processed using a

hash function, producing a value in the range of zero and one. A critical parameter, denoted as θ), is determined, which signifies the threshold for filtering these hashed values. The default value of theta is 1. When k values are inserted in the sketch theta is still 1. When the next unique value must be inserted into the sketch, theta gets the value of the (k+1)th minimum value and that value is removed from the cache. Only values below the theta threshold are considered, which helps in determining the cardinality of the set.

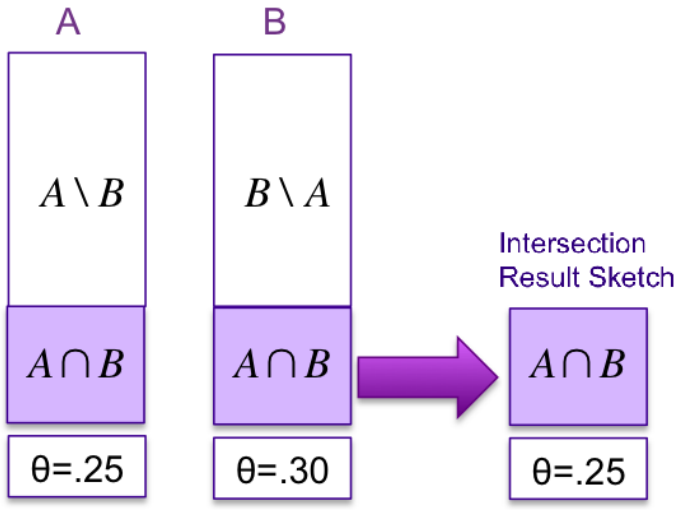


FIGURE 2.5: Example of an intersection operation between two Theta Sketches. Adapted from [2]

The elegance of Theta Sketches lies in their adaptability to various set operations. The combining function of this method is quite simple and easy to perform. When merging two sketches, the θ) value of the resultant sketch is the minimum of the θ) values of the original sketches. The set of hashed values of the input sketches are then filtered based on this new threshold, providing the set of hashed values of the new sketch. Similar principles apply for intersections and differences, with suitable modifications in the approach as explained by the following expressions:

$$\Delta = \{\cup, \cap\}; \theta_{\alpha\Delta\beta} = min(\theta_\alpha, \theta_\beta); S_{\alpha\Delta\beta} = \{x < \theta_{\alpha\Delta\beta}; x \in (S_\alpha \Delta S_\beta)\} \tag{2.1}$$

The third element of the framework, which is the estimation function, is also quite easy to implement. In order to calculate the cardinality of the input set, after the creation of the sketch one must divide the number of elements in the sketch with the theta value. Another notable aspect is the error bounds of Theta Sketches. The error in estimation is probabilistic, and its bounds can be explicitly defined based on the space allocated to the sketch. This provides users with a clear trade-off between accuracy and space, allowing for informed decisions based on specific application requirements. As mentioned by Dasgupta et al., the relative standard error of the proposed method is: $\frac{1}{\sqrt{k-1}}$, which is an improvement from the normal KMV sketch that has an error rate of: $\frac{1}{\sqrt{k-2}}$. A significant advantage of Theta Sketches is their space efficiency. While the memory consumption does increase with the desired accuracy, it's still markedly lower compared to deterministic algorithms for the same task. Furthermore, as with HyperLogLog, multiple Theta Sketches can be combined, which is invaluable in distributed computing environments where data might be scattered across different nodes.

However, it's worth noting that while Theta Sketches excel in approximating set operations, they are not intended for exact computations. This is a fundamental characteristic of probabilistic data structures, where a certain degree of approximation is accepted in favor of efficiency.

In conclusion, Theta Sketches have emerged as a robust solution for approximating set operations in the context of massive data streams. Their combination of space efficiency, adaptability to various operations, and clear error bounds positions them as a tool of choice for many real-time analytics and database systems as Theta Sketches provide a scalable and efficient solution.

2.2 Time series

Data streams, characterized by their multidimensionality, play a crucial role across diverse fields. Their temporal dimension means that patterns and sequences within the data are as essential as the data values. Unlike other data types, data streams require unique analysis techniques due to this multidimensional aspect. With the surge in the volume of data streams, especially in today's applications, the computational demands of similarity searches can become overwhelming. This has led to a surge in research efforts aimed at developing optimized indexing methods, techniques to reduce data dimensionality, and approximation strategies. The ultimate goal is to strike a balance between computational efficiency and search accuracy, ensuring that the results are both quick and precise.

The intricate nature of both time series and data streams, especially their temporal aspects, appears as an opportunity to transpose methods from time-series analysis into the realm of data streams. With a plethora of techniques in time-series similarity approximation available, it was a challenge to select the best fit. To this end, the exhaustive survey by Echihabi et al. [11] was referred to for guidance. This survey was particularly enlightening, offering detailed comparisons of a bunch of time-series approximation methods and supplementing them with experimental results. This comprehensive assessment granted a clearer understanding of each technique's potential and challenges, ensuring that the chosen approach was both resilient and apt for the research goals.

Echihabi et al. [11] in their survey cast a spotlight on seven standout techniques tailored for multidimensional data. Beyond this, the survey also elaborates on innovative approximate search algorithms, tailoring them exclusively to the unique demands of data series analysis. These algorithms were of particular interest to us for the purposes of this work so they will be emphasized, these algorithms are DSTree [34], iSAX2+ [6] and VA+file [13]. Notably, their evaluation was grounded in well-defined criteria: scalability, search efficiency, and accuracy. Their experimentation was methodically structured into two distinct phases: index building and query answering. Furthermore, the data-sets utilized were both synthetic and real, ensuring their results had breadth and applicability.

In evaluating the scalability of various indexing methods, distinct patterns and efficiencies emerge. As depicted in Figure 2.6, iSAX2+ stands out as the most expedient in index building. Following closely are VA+file, SRS, DSTree, FLANN, QALSH, IMI, and HNSW, in that order. Notably, despite IMI and HNSW being the sole parallel methods in this comparison, they lagged behind, exhibiting the slowest index-building speeds. Assessing the memory footprint of each method, as it can be

seen in the same Figure, DSTree emerged as the leader in efficiency, with iSAX2+ following. The other techniques lagged significantly, exhibiting inefficiencies of nearly two orders of magnitude.

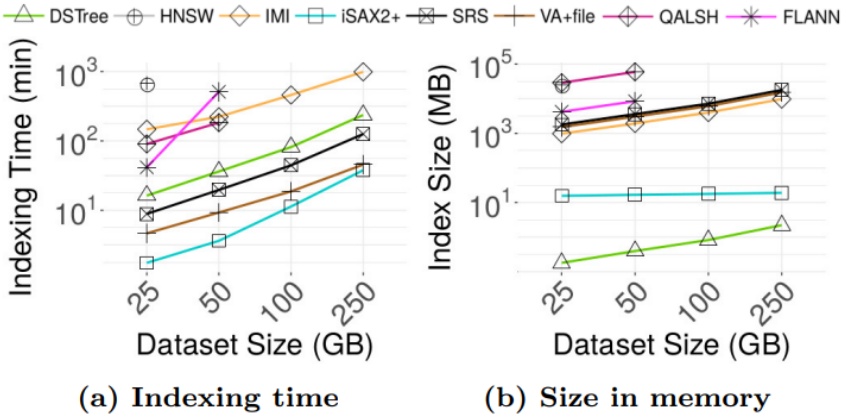


FIGURE 2.6: Comparison of indexing scalability for various techniques designed for analyzing time series data. Source: [11]

Shifting our focus to the evaluation of query answering efficiency and accuracy, DSTree and iSAX2+ consistently eclipse other techniques in performance. Examining the delicate balance between query efficiency and accuracy, DSTree consistently outperforms iSAX2+ across the majority of datasets. Although LSH techniques, typified by methods like SRS, offer assurances on the search result accuracy, they come at a steep price, consuming both considerable time and memory resources. Conclusively, in the landscape of approximate query answering, DSTree and iSAX2+ stand out, overshadowing even state-of-the-art LSH-based methods like SRS and QALSH in terms of both time and space efficiencies. Furthermore, they also offer more robust theoretical guarantees.

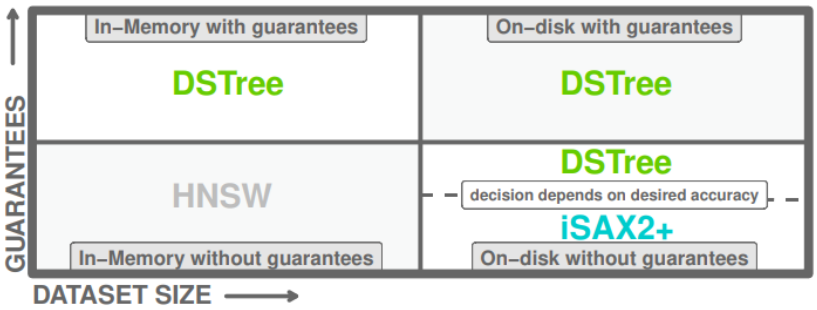


FIGURE 2.7: Results of various experiments to evaluate the accuracy of many time series analyzing techniques, regarding the accuracy when answering similarity queries both on disk and in memory. Source: [11]

2.2.1 DSTree

After, consulting the results of the aforementioned survey, the DSTree algorithm was chosen as the third method to calculate similarity between data streams. Even within this elite bracket of top-performing methods as presented in [11], DSTree establishes itself as the superior choice for a majority of scenarios. Recognizing this distinct

edge, DSTree was selected as the primary method for in-depth examination in this research. Its performance attributes undoubtedly render it a prime candidate for efficient similarity approximation in streaming environments.

The DSTree, an indexing structure, is meticulously crafted to manage large-scale data series datasets, balancing both storage optimization and query performance. Central to its operation is the concept of adaptive segmentation. Instead of the conventional fixed-length segmentation of the dataset as the tree grows in size and more levels are added, DSTree opts for a more refined technique. By employing pattern-based segmentation, it identifies natural breakpoints or changes in trends within the data. This leads to segments of variable lengths, which are more reflective of the underlying data patterns and ensure a highly representative indexing structure.

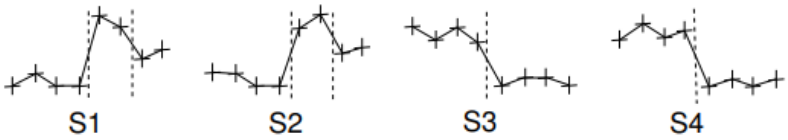


FIGURE 2.8: Example of dynamic segmentation. By applying this segmentation it will be easier to reduce the dimensionality of the time series and create more efficient indexes. Source: [34]

Inserting a time series into the DSTree involves directing it to an appropriate leaf node, aiming to group similar series together. Starting from the root node, if it's a leaf with empty room, the input is placed there; otherwise, the node gets split. If it is an internal node, the new time-series moves to the most fitting child node. This continues until the new time-series reaches a leaf. The insertion process relies heavily on the functions BestSplit() to determine the optimal node splitting strategy, and routeToChild() to decide on the child node. Both functions are detailed in [34].

Partitioning time series into subsets for node assignment in the DSTree can be accomplished in two primary ways: horizontal splitting (H-split) and vertical splitting (V-split). During an H-split, the segmentation remains consistent, but the time series set divides into two separate subsets. The series are categorized into these subsets based on specific criteria of a chosen segment - either its mean or its standard deviation.

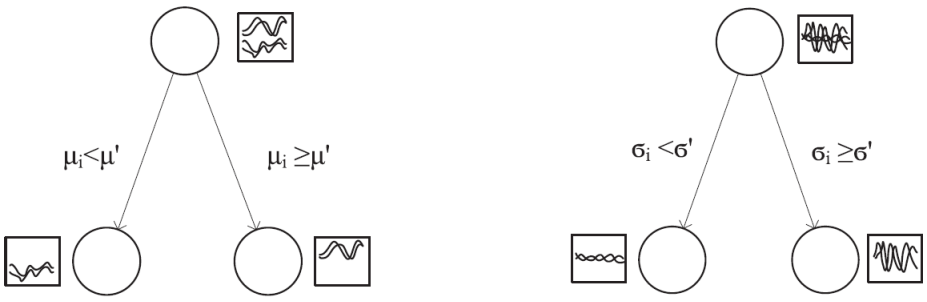


FIGURE 2.9: Examples of both cases of horizontal split.
Splitting using mean and splitting using standard deviation.
Source: [34]

On the other hand, V-split refines the current segmentation by adding an extra segment. As an example, consider Figure 2.10 where four time series share similar mean and standard deviation values for the i-th segment, making them difficult to distinguish through H-split. To address this, a V-split first bisects the segment, and then the series are clustered by the mean of the left sub-segment.

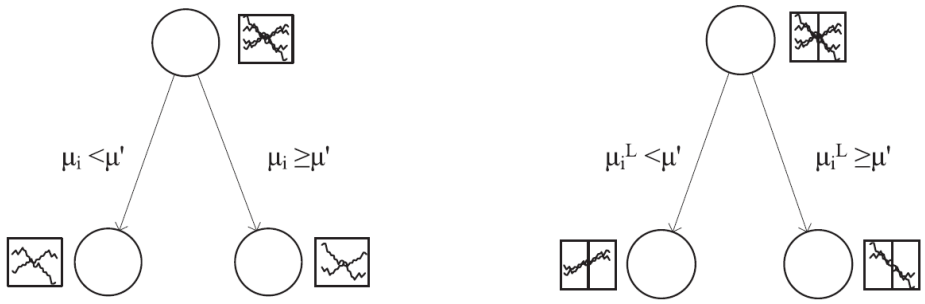


FIGURE 2.10: Comparison of splitting horizontal (left) and vertical split (right). Source: [34]

Finally, within the framework of the DSTree, two types of queries are catered to: traditional similarity search and the estimation of distance distribution. For the purposes of this research, the focus is squarely on the traditional similarity search, as it's most pertinent to the subject under investigation.

DSTree operates in a systematic, two-phase manner. The first step is to provide a Best-So-Far (BSF) answer. Given a query, this is achieved by traversing the tree in a similar way as if the query was to be inserted. Once a leaf node is reached the data in the leaf node are compared with the query and the minimum distance is calculated. After determining the BSF, a priority queue is formulated. This queue is tasked with scrutinizing nodes that could contain series more akin to the query than the BSF. Initially, this queue contains only the root node. The algorithm consistently pulls the node with the smallest minimum distance from the priority queue. This process continues until the priority queue is empty or an early termination criterion gets triggered. This termination arises when the minimal distance surpasses or matches the BSF's distance. When this happens, it's certain that no other series in the priority queue can outmatch the BSF in similarity, allowing for their exclusion from the search.

The DSTree, inherently developed for static data contexts, has not been directly tailored for handling streaming data. To transition it into this realm requires several substantive modifications, which will be analyzed in Chapter 3.3.3. Yet, preliminary assessments are promising. Initial data, even with the challenges of streaming environments in mind, suggest that the DSTree possesses potential. With proper adaptation, it could emerge as an efficient mechanism for evaluating the similarities that exist between rapidly flowing data streams.

Chapter 3

Methodology

Handling data streams presents distinct challenges that make reliance on traditional systems, like database management systems (DBMS), inadequate. The unbounded nature of data streams rendered these systems inappropriate for data analysis in streaming environments. Their fundamental architecture doesn't support rapid and continuous data ingestion, a vital requirement for streaming data. Beyond mere speed, data streams brought to the fore the significance of approximation and adaptivity in data processing. Unlike DBMSs, which prioritize delivering exact answers and operate based on stable, predefined queries, data streams demanded a shift in paradigm. In the context of streams, absolute precision often took a backseat. Researchers quickly realized that to effectively process and analyze streaming data, one needed systems that could provide nearly accurate results on-the-fly, adapt to changing data trends, and accommodate the continuous flow of incoming information.

In recent years, the field of stream processing has seen a surge of development, with many new frameworks emerging. With so many options available, it can be difficult to choose the right framework for a particular use case. To make the most informed decision, it is important to compare the different frameworks and consider their respective strengths and weaknesses.

3.1 Apache Flink

Apache Flink stands out as a premier, open-source platform primarily tailored for stateful processing of diverse data types, encompassing both real-time streams and more static batches. Its distinct edge lies in its capability to remain operational non-stop, swiftly processing incoming data. A significant portion of these computations is executed in-memory, resulting in enhanced speed and notably reduced data processing latency, compared to other data streams processing systems like Apache Hive [31].

Diving deeper into its features, Flink's advanced state management system is hard to overlook. It offers users a guarantee that each piece of data will influence the system only once. This "exactly-once" processing ensures impeccable data accuracy, a vital trait when confronting challenges like out-of-sequence or tardy data entries, scenarios Flink is adept at navigating. This is a feature that sets Flink apart from other frameworks like Apache Storm [32].

Another advantage of Flink is its unified approach. While it's primarily designed for stream processing, it also offers tools for batch processing, all under one unified programming model. This flexibility allows developers to work with various data types seamlessly. Another known framework that also offers a certain amount of flexibility is Apache Spark. While Flink excels in low-latency, high-throughput

stream processing, Spark is known for its fast batch processing capabilities. Both frameworks can process large volumes of data quickly, with Flink focusing on real-time analytics and Spark catering to batch data processing tasks. For the purposes of this work Apache Flink was the most preferred choice.

Flink is versatile in its application. It can power event-driven apps, real-time data analytics, and even replace traditional batch-based processes, offering faster data transformations. Its internal structure involves data flow graphs that outline how data is transformed, processed, and subsequently routed to its designated endpoint, ensuring a transparent and understandable trajectory of data movement.

In terms of reliability, Flink offers robust fault tolerance. It uses checkpoints to save application states at regular intervals, so if there's a system failure, minimal data is lost. This system is also flexible, allowing developers to make changes to their Flink job without losing the application's state. Additionally, its distributed nature makes it scalable. It can process data concurrently across many machines, ensuring it can handle large data volumes. Data and application states are distributed across nodes, which boosts performance by using local data for faster computations.

Furthermore, Flink has a number of proven connectors to popular messaging and streaming systems, data stores and search engines like Apache Kafka and Amazon Kinesis. It offers different programming levels, from high-level SQL streaming to detailed APIs, so developers can choose the best tools for their tasks.

In summary, Apache Flink's design and features make it a reliable choice for processing large-scale data in real-time and batches. It's because of all these strengths that Apache Flink was regarded as the best choice for implementing our solution.

3.2 Condor

As mentioned before, synopses are summaries of data that can be used to estimate the results of complex queries without having to process all of the data. This can lead to significant performance improvements for data stream processing applications. However, only a few existing stream processing frameworks support synopses as first-class citizens [23]. and more specifically many of the existing algorithms do not implement synopsis in a distributed fashion. This means that developers have to manually implement support for synopses in their applications, which can be time-consuming and error-prone. In order to tackle this issue, Poepsel-Lemaitre et al. developed the Condor framework [28]. It is, a framework that facilitates the definition of synopsis-based streaming jobs and integrates them into datafow systems that support window processing. Since Condor also provides an integration to Apache Flink, it considered as a potential tool in order to implement the probabilistic data structures described in the previous chapters. In this chapter an overview of Condor will be presented, highlighting the features that make it a useful tool for the purposes of this work.

The Condor Framework emerges as a robust solution for managing data stream analysis, particularly oriented towards real-time or near real-time analytical scenarios. Its cornerstone lies in the support for synopses, simplifying the specification and processing of synopsis-based streaming jobs, thereby potentially enhancing the efficiency and ease of conducting real-time data analysis. More specifically, a notable feature of Condor is its ability to represent synopses as a particular case of windowed aggregate functions [20], hence allowing it to abstract internal processing details. This abstraction is instrumental for individuals or teams keen on focusing

on analytical tasks, without getting entangled in the underlying mechanics of data processing.

There are three conditions that must be covered according to the Condor model. The unnecessary details of the internal processes must be hided in order to make the synopsis easy to use. Additionally, the level of abstraction should not be an obstacle for performing the computations in a distributed way, and in such a way that the throughput would scale linearly depending on the number of nodes. Finally, take into account the different algebraic properties of each category of synopsis and define the optimizations that can be introduced in each one or the restrictions that can be imposed.

The process followed by Condor consists of two parts, first is a categorization process where synopsis are classified based on their algebraic properties. Secondly, the exploitation of a divide and conquer strategy means that the framework can achieve the distribution of the necessary computations across multiple nodes. By distributing the computation, it not only facilitates high performance but also scales linearly with the increase in data volume and complexity, which is essential for managing large-scale data streams in real-time analytics.

According to the suggested model, there are five classes of synopses: mergeable, commutative, invertible, order-based, and non-mergeable. A synopsis is a mergeable one, when there is a function that can combine two of its instances without alterations to the error and size guarantees [1]. Commutative synopses are a subclass of the mergeable ones, with the addition of an update function that follows the commutative property. Given that feature, these synopses can accept out of order elements. Invertible synopses, are themselves a subclass of commutative synopses, with a function that reverts changes to a previous state. This property can be used in order to resolve inconsistent results by decreasing updates instead of recomputing whole windows. An ordered-based synopsis retains ordering properties for every input element. last but not least, non-mergeable synopses only have an update function and do not hold any of the aforementioned features. Obviously in these cases parallelism cannot be exploited and the synopses are computed in a centralized way.

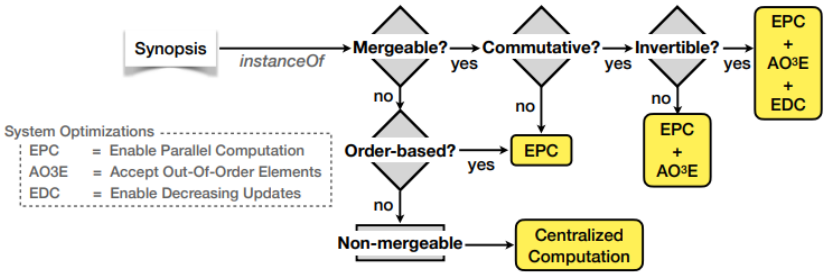


FIGURE 3.1: Synopses classification and possible optimizations for each category. Source: [28]

Furthermore, the process of translating a synopsis to an aggregate window function can be divided in three stages, divide, compute and merge. In the divide phase, the input data stream is distributed across all available cores, with each window of data being partitioned into multiple smaller windows to ensure a balanced distribution. Transitioning to the compute phase, the emphasis is on incrementally calculating partial aggregates. A partial synopsis is initiated and continuously updated as new data elements arrive. This ongoing update mechanism is essential for maintaining an accurate representation of the data. The compute phase reaches its conclusion

once all elements within the corresponding window have been incorporated into the synopsis, signifying the readiness for the final stage. The culmination occurs in the merge phase, where all the partial synopses are consolidated into a single result for every window. A merge function is employed to combine these partial synopses, ensuring a precise and comprehensive aggregation of the data.

Condor establishes the mathematical groundwork for efficient synopses computation in distributed streaming applications, ensuring high-throughput in parallel synopsis maintenance without compromising accuracy. Its performance scales linearly with the parallelism degree, making it a reliable framework for real-time analytics in distributed environments.

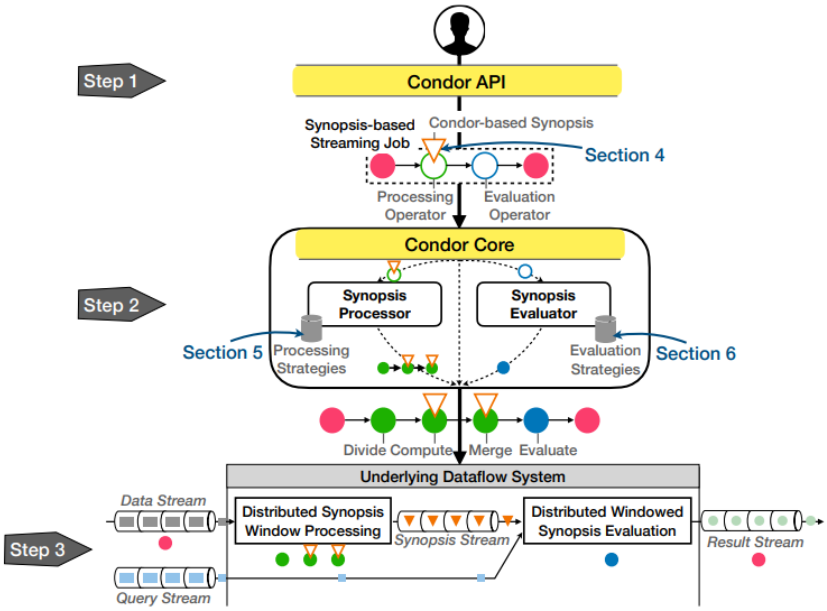


FIGURE 3.2: Detailed overview of Condor's architecture. Source: [28]

Through this well-structured process, HyperLogLog transcends its primary role of cardinality estimation to become an instrumental component in a robust methodology for similarity calculation.

3.3 Similarity Estimation Calculation

This section showcases the details of our contribution. In the earlier discussions, we talked about using Apache Flink to handle streaming data, and Condor for working with probabilistic data structures. To do this, we made some additions to the Condor framework by including two more synopsis methods. While Condor already has twelve synopsis algorithms, including HyperLogLog, the list was expanded with the addition of Theta Sketches and MetaDataSketch. MetaDataSketch is a custom made synopsis that will be used for the implementation of DSTree, as it will be descried further in Chapter 3.3.3.

However, yet, a crucial step remains - figuring out the similarity of data coming from different streams. To enhance the processing and analysis of streaming data, a structured approach was adopted in organizing the code. A naming convention was introduced to maintain consistency, where each class name begins with the name of the data structure it relates to, hereinafter referred to as 'X'. For example two such classes are HLLBuilder and ThetaBuilder, which are part of the HyperLogLog and

Theta Sketch implementation respectively. Java was the chosen language for this endeavor, as it is one of the two languages supported by the Flink framework and being the language in which Condor was originally written. This choice ensured a smooth integration with the existing frameworks.

For each selected data structure, a uniform structure was adhered to. Five main classes were created for each data structure to ensure the required functionality. The first class, named XSketch, was crafted based on the guidelines presented by Condor's development team in a demo that can be found at the project's GitHub page. This class is instrumental in defining the data structure and its variables, alongside housing a merge and an update function, as dictated by Condor's commutative synopsis interface. These functions are essential for managing the data and ensuring it's processed accurately.

The second class, dubbed XBuilder, plays a pivotal role in creating a build configuration for the corresponding data structure. It takes the input data stream and the window size as arguments in its constructor, laying down the foundation for processing the streaming data. The incoming streams can be of two different types, either read from a .txt file or arriving from a Kafka queue. The utilization of Apache Kafka will be analyzed further in Chapter 4. Embedded within XBuilder is a function designed to read the incoming data from the input stream, create a synopsis for each window, and consequently return a stream of synopses. This function acts as a conduit through which the raw streaming data is transformed into a structured synopsis, ready for further analysis.

Having mapped each incoming stream to a stream of probabilistic data structures, the subsequent steps are focused on calculating the similarity between these incoming streams using additional classes. A function named XSimilarity is introduced, which is fairly straightforward— it houses a function that takes two sketches as input and returns the measure of their similarity. Before analyzing the other function let's add some context. The way the data will be compared in our solution is quite simple and straight forward. here are two streams of data in play: one stream of data arriving from a Kafka queue, and a parallel stream of queries arriving from a second queue. Both streams are segmented into windows of identical size, and each window from the query stream is set to be compared with every window from the data stream received until that point. The previously explained code is utilized to map both streams to two streams of synopses, laying down the groundwork for similarity computation.

The final class is called XSimilarityComparator and extends Flink's RichCoFlatMap class, implying a common state shared between the two streams, which is a critical feature for managing and comparing the data in sync. The initialization of this state is carried out using the "open" function. Post initialization, as synopses arrive from the query stream, they are appended into a list. Subsequently, whenever a synopsis arrives from the data stream, it's compared against all the synopses present in the list until that point. The most similar synopsis is identified, and the result is encapsulated in a string. This string comprises the IDs of the two compared synopses alongside their similarity percentage, providing a succinct yet insightful representation of the similarity between the data streams at that instance.

So, through this setup, we have a straightforward way to compare our data streams, find out how similar they are, and get a clear output of the results. In the following chapters we will present in greater detail how the aforementioned process is implemented for every data structure and . We'll unfold the intricacies of the implementation, shedding light on the most engaging and crucial parts of the process.

3.3.1 HypeLogLog

Mapping the strings of incoming data to streams of synopses was quite simple in this case, since Condor by default comes with an implementation of the HLL sketch. This favorable setup allowed the focus to shift towards the more critical aspect of computing the similarity between the newly formed data streams.

HLL is traditionally known for its prowess in approximating the cardinality, essentially quantifying the number of distinct elements within a dataset. While it's not originally devised for similarity calculations between datasets, HyperLogLog facilitates the straightforward calculation of the cardinality of the union of sets. To obtain the count of the union of two sets, the HLL data structures representing the two sets can be 'merged' to form a new HLL data structure, from which the count can be derived. The merge operation for two HLLs, provided they have an equal number of buckets, entails comparing each pair of buckets and assigning the maximum value from each pair to the corresponding bucket in the resultant HLL. Building on this, the inclusion-exclusion principle comes into play to estimate the cardinality of the intersection indirectly, using the cardinalities of the individual sets and the cardinality of their union according to the following formula:

$$|A \cap B| = |A| + |B| + |A \cup B|$$

This phase harnesses the formula of the inclusion-exclusion principle to deduce the size of the intersection. With the size of the intersection and the size of the union at hand, the Jaccard Similarity is computed by dividing the size of the intersection by the size of the union.

But as mentioned and proved in [9] this approach introduces extra error and latency. More specifically, while HyperLogLog (HLL) is proficient at estimating the cardinality of union of sets, it does face challenges with other set operations, especially when the sets have significantly different sizes. When the size of the intersection is much smaller than the size of the union , HLL's error rate can increase, which negatively impacts the accuracy of the estimates. The error scaling behavior of HLL in such scenarios can overwhelm the inherent accuracy advantages of this method.

While trying to find alternative methods to tackle this issue we came several methods that use HLL. Prominent among these are an enhanced version of HLL, dubbed HyperLogLog++ (HLL++) [16] and, maximum-likelihood-based methods such as Maximum Likelihood Estimator (MLE), Joint Maximum Likelihood Estimator (JMLE), and Ertl's Improved method [12]. The comparison of these methods to each other, as well as to the inclusion-exclusion approach, was presented by Baker and Langmead [3].

In some initial tests, both HyperLogLog++ and Ertl's Improved method didn't do as well as MLE, so they were not considered further for that analysis. Additionally, Ertl's Joint MLE method, despite its superior accuracy, falls short due to the substantial latency it introduces, making it unsuitable for real-time data stream applications where speed is of the essence. On the other hand, while MLE exhibited promising results in a static setting, its computationally intensive nature posed a significant obstacle in data stream contexts. The method demands complex calculations involving exponentiations, divisions, harmonic means, and iterative procedures for finding roots of functions. This computational complexity, although manageable in a static environment, becomes a prohibitive factor in the dynamic, fast-paced realm of data streams. Also it is a much more complicated algorithm than the one based

on the inclusion-exclusion principle without showing any significant improvement in accuracy to justify the extra effort required to implement it.

For these reasons we decided to include the first approach, the one that is based on the inclusion-exclusion property. In order to improve the accuracy of this method we expanded the implementation provided by Condor by adding the following functionality: The use of multiple hash functions can potentially lead to better distribution and representation of the data across the buckets in the HyperLogLog data structure. Different hash functions may produce different hash values for the same data, which could lead to a more uniform distribution of data across the buckets. This, in turn, could help in reducing the variance and, subsequently, the error rate in the cardinality estimation. Averaging these estimates will lead to a more accurate and reliable estimate.

Finally, when configuring the HLL sketch, the user can configure the number of registers as well as the number of hash functions that will be used.

3.3.2 Theta Sketches

Integrating a new data structure like theta sketches into Condor necessitates an extension of the framework. When a user brings in a new synopsis into the system, it is crucial to manually categorize it into the right class so that Condor can recognize its algebraic properties, ensuring appropriate handling within the framework. Condor's primary objective is to offer a framework that facilitates the specification of synopsis-based streaming jobs in a broad sense. Therefore, the precise implementation of crucial operations like merge and update functions for each synopsis is dependent on the user. This user-dependent nature allows for the flexibility and customization necessary to cater to various data processing and analysis requirements, making Condor a versatile framework for managing streaming data.

In our case, Theta Sketches are part of the Commutative Synopses category, as declared by Condor, and thus, extends the CommutativeSynopsis class. This means that an update and a merge function must be implemented. By defining how new data is accommodated and how data fragments across windows are unified, these functions play a critical role in enabling the efficient and accurate processing of streaming data, as explained in Chapter 3.2.

Implementing the functions for integrating Theta Sketches into the Condor framework was fairly straightforward. Based on the work laid down by the creators of Theta Sketches, Dasgupta et al., in the paper [9]. This work provided a solid base and was expanded by the Apache Datasketch [30]. For a more hands-on understanding and practical insights into the implementation, the Apache DataSketches documentation was a valuable resource. The Apache DataSketches documentation, while based on the aforementioned paper, goes a step further. It provides a deeper exploration of the mathematical concepts that underlie Theta Sketches, making it easier to understand the theory behind the method. At the same time, it also offers a more abstract, high-level overview of Theta Sketches, making the method more accessible.

The combination of the theoretical grounding from [9] and the practical insights from the Apache DataSketches documentation facilitated a well-rounded understanding, aiding in a smooth implementation of the necessary functions within the Condor framework. This balanced approach ensured that the integration of Theta Sketches was not only accurate but also well-informed, bridging the gap between theory and practical application effectively.

following the definition of the sketch, the remaining process adheres to the steps and flow outlined in the previous chapter. The choice of Theta Sketch was driven by its adeptness in handling set operations, which significantly simplified the task when compared to the effort required with HyperLogLog (HLL) for calculating Jaccard similarity between datasets. Unlike HLL, where additional efforts were necessary to maneuver through set operations for similarity calculations, Theta Sketch offered a more direct and less cumbersome approach. Theta Sketch's inherent capability to manage set operations provided a more straightforward pathway to implement a solution for computing the Jaccardian similarity.

3.3.3 DSTree

When implementing a new feature or algorithm like DSTree within a framework like Condor, a structured approach to coding is essential. The general outline of how the code structure was organized is similar to the one described in the previous chapters to ensure clarity, maintainability, and the seamless integration of the new feature. However, the unique functionalities brought by DSTree necessitated the creation of a new probabilistic data structure since none existed that could cater to these specific functionalities.

MetaDataSketch

The synopsis for each window is envisioned as a data structure encapsulating two 2-dimensional matrices. One of these matrices stores the mean values of each segment and its sub-segments, while the other matrix houses the corresponding standard deviation values. This dual-matrix structure serves as a compact, organized representation of the necessary statistical data, facilitating the DSTree's ability to execute its operations effectively. Additionally, a third array of the same size is required, holding the count of elements that arrived in each corresponding sub-segment of the window. This array is required only for the update and merge functions and ca then be discarded once the synopsis is created, in order to free memory.

For example, given a window of 32 elements and a predefined segmentation of 8 elements, the requirement was to ascertain the mean values across different segment sizes—every 8 elements, every 16 elements, and the entire window of 32 elements. At the first level of this hierarchy, the mean value of the entire window of 32 elements will be computed and stored in the first row of a designated matrix. Proceeding to the second level, the window will be separated in two halves, each consisting of 16 elements. The mean values of these two halves will be computed separately and stored in the second row of the matrix, with the mean of the first 16 elements occupying the first position, and the mean of the latter 16 elements occupying the second position. The third row will require the window to be divided into four quarters, each containing the mean values of 8 elements. Simultaneously, a similar process is carried out to compute and store the standard deviation values of each segment and sub-segment, utilizing a separate matrix structured identically to the mean values matrix.

Since MetaDataSketch implements the CommutativeSynopsis interface, an update and a merge function must be implemented. The update function is crucial for recalculating the mean and standard deviation values as new data elements arrive in the stream. However, a key challenge emerges due to the streaming nature of the data - typically, the calculation of mean and standard deviation requires the entire

dataset to be available upfront, but in this scenario, data elements are arriving incrementally. To navigate this challenge, an incremental approach is adopted to adjust the average and standard deviation with each new data element that arrives [22]. The objective is to compute the cumulative average seamlessly as the data streams in. This goal is facilitated by the following formula which is geared towards calculating the cumulative average:

$$CA_{n+1} = CA_n + \frac{x_{n+1} + CA_n}{n+1}$$

This formula requires only the knowledge of the value of the new element, the mean value of the elements received so far and the count of these elements, information which is readily available from the previously mentioned arrays.

The incremental calculation of the standard deviation is a bit more complicated, in the sense that it requires the calculation of an incremental value after every new element's arrival [35]. These values are kept in a fourth array of the same size as the mean values and standard deviation array. However, it is only used in the update function and can be discarded after the synopsis of the window is complete in order to make the algorithm more memory efficient. The formula for the calculation of standard deviation, with only one pass over the data, is the following:

$$s_n^2 = \sqrt{\frac{inc}{n}}$$

Where increment equals:

$$inc = \sum_{i=1}^{\infty} (x_n - \mu_n) * (x_n - \mu_{n-1})$$

Finally, the purpose of the merge function is to combine the count, mean and standard deviation arrays. In order to do that, the following equations are exploited and executed per tuple of each array in order to compute the aggregate statistics and combine the data from the different groups.

The formula for calculating the weighted mean (W) of two sets with known means (M1 and M2), the number of elements (N1 and N2), and the total number of elements in both sets (N1+N2) is as follows:

$$W = \frac{N1 \cdot M1 + N2 \cdot M2}{N1 + N2}$$

The formula for calculating the combined standard deviation (SS) of two sets with known standard deviations (S1S1 and S2S2), means (M1M1 and M2M2), the number of elements (N1N1 and N2N2), and the total number of elements in both sets (N1+N2N1+N2) is as follows:

$$S = \sqrt{\frac{(N1-1) \cdot S1^2 + (N2-1) \cdot S2^2 + N1 \cdot (M1-W)^2 + N2 \cdot (M2-W)^2}{N1+N2}}$$

DSTreeNode Append

In adapting the DSTree data structure for streaming environments within the Condor framework, a novel data structure named DSTreeNode has been introduced. Drawing inspiration from the original DSTree as expounded in [34], it incorporates

modifications essential for functionality in a streaming scenario. The DSTreeNode encapsulates various variables akin to those described in the original paper.

One of the pivotal variables is 'C', representing the count of time series indexed in the subtree rooted at the node in question. An array of coordinates, dubbed 'sg', plays a crucial role in mapping positions within the meanValues and stdDevValues arrays of a MetaDataSketch synopsis, acting as a bridge to the statistical data. Another array, named 'z' exists ,where each row stores the minimum and maximum mean value and standard deviation of a different segment. The boolean variable 'leaf' serves as a flag, distinguishing between internal and leaf nodes

Diving deeper into the anatomy of an internal node, it houses two pointers steering towards its child nodes, thereby crafting the hierarchical scaffold of the tree. Each internal node also harbors a splitting strategy, denoted as 'SP', which orchestrates the logic for branching down to child nodes. The 'splitSegment' variable pinpoints the position of the segment that was split in the coordinated array sg, acting as a marker for the segment under scrutiny during the splitting decision. Complementing this, the splitting value acts as a threshold, aiding in making informed branching decisions. The mechanism of appending a new element to the Streaming DSTree is dictated by a confluence of the splitting strategy, split segment, and splitting value. For instance, with a splitting strategy of horizontal mean as described in 2.2.1, a split segment of 1, and a splitting value of 10, the algorithm evaluates the mean value of the first segment of the new element. A mean value less than 10 ushers the traversal towards the left child, while a value exceeding 10 directs it to the right child.

On the flip side, leaf nodes are repositories for MetaDataSketch synopses, with a defined capacity marking the threshold of synopses they can harbor before a split is necessitated. Hence, they contain an array of MetaDataSkethes named 'timeSeriesMetaData', as well as a variable named capacity and another named currentElements, which are both self explanatory.

Furthermore, the DSTreeNode class is equipped with a variety of functions to ensure it operates as required within the Condor framework. A crucial function among these is the 'append' function, which is triggered every time a new MetaDataSketch is to be inserted into the tree. When a new MetaDataSketch is added, the first thing that happens is an update to a value known as 'c', which keeps count of the number of time series indexed in the tree, incrementing it by one. If the node is not a leaf node, a function called 'findTestValue' is initiated first. This function checks a specific splitting strategy to decide which value from the new data will be compared with a predefined splitting value. Depending on the outcome of this comparison, the 'append' function is called again on one of the child nodes, ensuring the new data is placed in the correct part of the tree.

On the other hand, if the node where the new data is to be added is a leaf node, the process is fairly straightforward. The new element can be added to this node as long as there's room. However, if the node is already full, it needs to be split to create more space for the new element. For that purpose the split function is called. The splitting process is a bit more complex.

Firstly, another function called 'bestSplit' plays a crucial role when a node reaches its capacity and needs to be split to accommodate more MetaDataSketches. As the name 'bestSplit' suggests, this function's job is to determine the best way to split the node. Specifically, it aims to find the optimal segment and the optimal splitting strategy for the node. To find the best split, the function evaluates the benefit of splitting for every possible strategy and every possible segment within the node. This involves a double loop structure where the outer loop cycles through different splitting strategies, and the inner loop goes through different segments. The process

Algorithm 1 $N.Insert(X)$: N is a node, X is a time series

update Z in node N according to X;
if N is a leaf node **then**
  if $C < \psi$ **then** ▷ N has space to hold X
    Append X to data file pointed by N, $C = C + 1$;
  else ▷ $C == \psi$, no space in N to hold X
    Append X to data file pointed by N, $C = C + 1$;
    $SP = BestSplit()$;
    Create two children nodes for N;
    for each time series Y in N **do**
      $N' = N.routeToChild(Y, SP)$; $N'.insert(Y)$;
    end for
  end if
else
  $N' = N.routeToChild(X, SP)$; $N'.insert(X)$;
end if

FIGURE 3.3: The algorithm of adding a new time series in the DSTree, which was used as the base for the streaming version of the algorithm. Source: [34]

begins by calculating a splitting value for each combination of strategy and segment, by calling the 'findTestValue' that we saw earlier. Based on this splitting value, it's determined whether the new synopsis would go to the left or the right child of the node if a split were to happen at that point. The benefit of each possible split is then estimated using the following formulas, as described in the original paper that introduced this method.:

$$B = Q_{os} - \frac{Q_{os_l} + Q_{os_r}}{2}$$

Where:

$$Q_os = \sum_{i=1}^{m}((\mu_i^{max} - \mu_i^{min})^2 + (\sigma_i^{max})^2)$$

After calculating the benefit 'B' for each case, the function compares all the calculated benefits. The combination of strategy and segment that yields the highest benefit is considered the best way to split the node. The function then updates the variables 'splitValue', 'splitSegment', and 'SP' with the values from the best split found.

Once the best splitting strategy and the segment are identified through the 'bestSplit' function, it's time to execute the split. The splitting could be done in two ways: horizontally or vertically, each having different implications for the node structure. In a horizontal split, the total number of segments within the nodes remains unchanged. This type of split essentially redistributes the existing MetaDataSketches between the current node and a new node, based on the determined splitting value. The segments themselves remain as they were, maintaining their original structure. On the other hand, a vertical split alters the structure of the segments. In this case, the segment identified for splitting is actually divided into two sub-segments. In the child nodes resulting from this split, the tuple that initially held the information of the original segment is updated. Now, it holds the coordinates of the left half of the

split segment. Additionally, a new tuple is introduced to hold the coordinates of the right half of the split segment.

This adjustment in the structure allows the child nodes to have more detailed segmentation, which can potentially lead to more accurate organization and analysis of the data contained within them. The vertical split, by creating new sub-segments, offers a finer granularity in organizing the data, which could be beneficial in managing and analyzing the streaming data in the tree structure.

DSTreeNode Distance Calculation

The DSTreeNode class also encompasses a critical functionality of identifying the MetaDataSketch from the node's subtree that holds the most resemblance to a given MetaDataSketch query, along with returning the similarity value. This process of similarity determination, as highlighted in Chapter 2.2.1, is bifurcated into two primary steps, mirrored in the code for the streaming version through the invocation of the 'leastDistance' function, with the query being passed as an input.

The initial step within this function triggers the call to another function dubbed 'heuristicDistance.' The role of 'heuristicDistance' is to traverse through the tree and pinpoint the leaf node where the query would have been placed if it were appended to the tree. Afterwards, 'heuristicDistance' in turn calls a function named 'nodeDistance. The 'nodeDistance' function is tasked with computing the distance between the query and the closest MetaDataSketch residing in the node, employing the following formula:

$$D(X,Y) \geq \sqrt{\sum_{i=1}^{m}(\mu_i^X - \mu_i^Y)^2 + (\sigma_i^X - \sigma_i^Y)^2}$$

Interestingly, the formula provided in the original paper comprises two separate formulas capable of yielding the lower and upper bounds of distance. The choice was made to utilize the lower bound as the preferred metric for distance measurement. Post execution, 'heuristicDistance' furnishes the ID of the synopsis that, up to that point, holds the most resemblance to the query, along with an approximate value of the distance from the query.

Following the identification of the Best-So-Far (BSF) similarity, the next step entails the creation of a priority queue. This queue is instrumental in examining nodes that might house series more similar to the query. Initially, this queue is populated solely with the root node, setting the stage for further examinations down the tree structure.

To populate this queue effectively, the 'groupDistanceCalc' function is employed. This function utilizes the following formula, to estimate the minimum distance between the query and the contents of the subtree originating from the examined node. The essence of this formula is to quickly prune parts of the tree that are definetely less similar to the query than the BSF.

$$D_{min} \geq \sqrt{\sum_{i=1}^{m}(LB_i^{\mu} + LB_i^{\sigma})}$$

Where:

$$LB_i^{\mu} = (\mu_i^{min} - \mu_i^X)^2 \text{ , if } \mu_i^X \leq \mu_i^{min}$$

$$LB_i^{\mu} = 0 \text{ , if } \mu_i^{min} \leq \mu_i^X \leq \mu_i^{max}$$
$$LB_i^{\mu} = (\mu_i^{max} - \mu_i^X)^2, if \mu_i^{max} \leq \mu_i^X$$

$$LB_i^{\sigma} = (\sigma_i^{min} - \sigma_i^X)^2 \text{ , if } \mu_i^X \leq \sigma_i^{min}$$
$$LB_i^{\mu} = 0 \text{ , if } \sigma_i^{min} \leq \sigma_i^X \leq \sigma_i^{max}$$
$$LB_i^{\sigma} = (\sigma_i^{max} - \sigma_i^X)^2, if \sigma_i^{max} \leq \sigma_i^X$$

If the returned value is lesser than the BSF, it indicates the potential presence of a more similar synopsis within that node. Consequently, this node is added to the priority queue for further scrutiny. However, if the value is greater, it implies that the node under consideration is unlikely to contain a more similar synopsis, thus the process moves on to the next node without adding the current node to the queue.

Upon reaching a leaf node during the traversal process, the 'nodeDistance' function is once again called into action to scrutinize the similarity between the contents of this leaf node and the query. Unlike internal nodes, leaf nodes house actual MetaDataSketches, making them the eventual targets for similarity examination. If during this examination, a MetaDataSketch is found that bears a closer resemblance to the query than the previously identified Best-So-Far (BSF) similarity, then the BSF is updated to reflect this newfound similarity. This ensures that the process is continually honing in on the most similar MetaDataSketch as it navigates through the tree structure.

Algorithm 2 $exactSearch(Q)$

Input: A query time series Q
Output: The nearest time series TS with distance D_{bsf}
$N_{bsf} = HeuristicSearch(Q)$;
$(TS, D_{bsf}) = calcMinDist(N_{bsf}, Q)$;
Initialize distance priority queue pq;
$pq.Add(N_R, D_{LB}(N_R, Q))$;
while $!pq.isEmpty()$ **do**
    $(N_{cur}, LB_{cur}) = pq.PopMin()$;
    if $LB_{cur} > D_{bsf}$ **then**
        break;
    end if
    if N_{cur} is a leaf node **then**
        $(X, Dist) = calcMinDist(N_{cur}, Q)$;
        if $Dist < D_{bsf}$ **then**
            $D_{bsf} = Dist; TS = X$;
        end if
    else
        for all children nodes N' of N_{cur} **do**
            if $D_{LB}(N', Q) < D_{bsf}$ **then**
                $pq.add(N', D_{LB}(N', Q))$;
            end if
        end for
    end if
end while
return TS,D_{bsf};

FIGURE 3.4: The algorithm used by the static version of DSTree in order to determine the most similar time series to a given query. Source: [34]

Chapter 4

Experiments and Evaluation

In this chapter, the metrics and the process designated for evaluating approximate nearest neighbor search techniques within a streaming environment are thoroughly described. These metrics are broadly classified into two primary categories: efficiency and accuracy. The efficiency metrics are designed to gauge the system's performance in terms of resource utilization and time efficiency. In contrast, accuracy metrics are focused on assessing the correctness and quality of the similarity search outcomes, ensuring that the results are reliable and reflective of the actual data relationships.

Moreover, to ensure a fair and comprehensive assessment, all the methods under scrutiny were evaluated within the same framework through a meticulously designed series of experiments. The experimentation process was methodically bifurcated into two discernible phases: index building and query answering. The index building phase focuses on the efficiency of constructing the necessary data structures, which is crucial for timely data processing. The query answering phase, on the other hand, zeroes in on the accuracy and efficiency of retrieving relevant data in response to queries, which is central to the utility of the system in real-world scenarios.

Furthermore, the experimentation was conducted with the use of two different kinds of synthetic data-sets. This dual-dataset approach bolsters the breadth and applicability of our findings, ensuring the results have a broader relevance and applicability. The synthetic datasets allow for a controlled evaluation environment, isolating and analyzing the impact of varying data characteristics.

In essence, this structured and encompassing approach to evaluation lays a robust foundation for dissecting the performance and reliability of different approximate nearest neighbor search techniques. It paves the way for a thorough understanding and analysis in the subsequent chapters, where the nuances of each method and the implications of their performance metrics will be delved into in greater detail.

4.1 Set Up

We use our custom implementations of each method, HLL ThetaSketch and DSTree. All methods are single core implementations.

When carrying out experiments on a personal laptop, variables such as unexpected background processes or system updates can affect performance, leading to inconsistent results. To address this, each test was repeated three times, and the reported outcomes represent the mean of these trials. Nevertheless, these external factors mean that the specific numerical results should be taken with a grain of salt.

Instead, the focus should be on the system's behavior and the comparative performance under different conditions, rather than the precise numbers themselves. This approach prioritizes understanding the relative impact of changes in the system's configuration over the exact figures recorded during the testing phase.

It is also worth mentioning that the absolute numbers of execution time and throughput that will be presented are highly affected by the system's specifications and they would differ significantly if the same code was executed in an alternative environment.

4.2 Data Sets

Two different datasets were used in the experiments that were conducted in order to evaluate the aforementioned methods. The first was a set of time series created by a research team in order to help researchers that are working on the area of similarity search.

The second was created specifically for the purposes of this project. More details will be presented in this section.

4.2.1 Dataset for Approximate Similarity Search

In their contribution [19], Laurent Amsaleg and Hervé Jégou introduce a comprehensive set of evaluation datasets to assess the performance of approximate nearest neighbors search algorithms across various data types and database sizes. Each dataset is meticulously structured into three subsets of vectors: base vectors, where the search is executed; query vectors, which are used to pose search queries; and learning vectors, employed to fine-tune the parameters intrinsic to a particular method. Additionally, they provide a ground truth for each dataset, encapsulated in the form of pre-computed k nearest neighbors based on their square Euclidean distance. This ground truth serves as a benchmark against which the accuracy and of the search algorithm under test can be measured.

The four datasets created vary in size and complexity, with three of them having a size of 128, but differing in the number of base and query vectors they encompass. Specifically, they contain 10K, 1M, and 1B base vectors and 100, 10K, and 10K query vectors respectively. The fourth dataset consists of larger vectors with a size dimension of 960, encompassing 1M base vectors and 1K query vectors. This gradation in dataset size provides a platform to evaluate how well the approximate nearest neighbor search algorithms scale with the size of the database, a critical aspect for real-world applicability.

The datasets are stored in three different file formats to cater to different data types: .bvecs or .fvecs for vector files and .ivecs for the ground truth file. To enhance usability, a Matlab script was crafted to convert these datasets into .txt file format, which is generally easier to work with and can be used as input to a Kafka queue.

For the experimentation phase of this study, only the smallest dataset, the one with 10K base vectors, was utilized due to the constraints posed by the hardware resources. The experiments were carried out on a single laptop which had limited memory and computational power, making it impractical to work with larger datasets. This decision was driven by the necessity to adhere to the available computational resources while still being able to conduct meaningful experiments to evaluate the approximate nearest neighbor search techniques in a streaming environment. Even with this limitation, valuable insights could be gleaned from the

experimentation on the chosen dataset, setting a foundation for further exploration on more robust computational setups in the future.

4.2.2 Custom Data Set

Given the limitations of the dataset provided by Laurent Amsaleg and Hervé Jégou, particularly in terms of the fixed window size of 128 and a maximum of 10,000 elements, a need arose to have a more flexible dataset for conducting a range of experiments. To address this, an API was developed to generate synthetic datasets where various parameters could be adjusted according to the experiment's requirements. Users have the flexibility to modify the window size, the number of windows to be created, and the number of queries to be generated. Additionally, they have the option to choose the nature of the data – it could either be serial data beginning from a specific number and incrementing by one, data derived from a predefined array, or randomly generated data following a normal distribution. In the latter case, users also have the control to adjust the mean value and the standard deviation to suit their experiment conditions.

Once the data is generated, it's dispatched to two separate Kafka topics. The primary topic houses the bulk of the elements and serves as the main data stream, while the secondary topic contains the queries that are to be compared against the elements from the first stream.

For evaluating the outcomes of the experiments utilizing this synthetic data, a program was developed to compare every query with the elements in the main data stream, identifying the 100 nearest neighbors for each query. This is achieved through a brute force approach to ensure accuracy in the results. The outcomes are then recorded in a text file, formatted similarly to the dataset by Laurent Amsaleg and Hervé Jégou, thus maintaining a consistent structure for easy analysis and comparison. This setup not only provides a platform for a variety of experiments but also ensures that the results are structured in a manner conducive for thorough evaluation and analysis.

4.3 Efficiency Evaluation

Efficiency is a critical aspect in evaluating similarity search techniques, especially in a streaming environment where timely data processing is essential. In our experiments, we aim to thoroughly assess the performance and resource utilization of these techniques using several metrics.

One of these metrics is Response Time, which measures the duration the system takes to process a query and deliver the results, crucial in real-time applications where delays could be detrimental. In our experiments, response time will be scrutinized in two distinct scenarios: during the creation of a synopsis, which involves measuring the time taken to summarize the data stream into a compact form, and during query execution, which involves gauging the time taken to compare an incoming query against all stored synopses to find the most similar one.

Another metric is Throughput, which assesses the number of records the system can handle per millisecond, serving as an indicator of the system's capacity and processing speed. Like response time, throughput will be evaluated in two key scenarios within our experiments: synopsis creation, assessing the rate with which the

system can create synopses from the incoming records, and query execution, determining the number of queries the system can process against the stored synopses per second.

Lastly, scalability is considered, which measures a system's ability to handle increasing data volumes efficiently. In our experiments, we aim to measure scalability by observing any changes in response time or throughput as the workload increases. A system that maintains a steady performance or shows only a slight performance decline with an increased workload is considered more scalable.

These metrics are designed to provide a structured approach to compare different similarity search techniques in a streaming context. By analyzing response time, throughput, and scalability, we aim to gain insights into each technique's performance and its potential to handle escalating workloads efficiently. This rigorous evaluation is indispensable for determining the most suitable similarity search techniques for real-world streaming applications.

4.3.1 Response Time

In this section, we will delve into the time analysis of each method in two significant aspects: the synopsis creation and query answering. The duration it takes for a method to generate a new synopsis and the time it requires to identify the most similar synopsis when given a query synopsis are fundamental metrics for evaluating the performance of the methods. The results will provide valuable insights into the suitability of each method for different streaming data scenarios and the trade-off between synopsis creation time and query answering time.

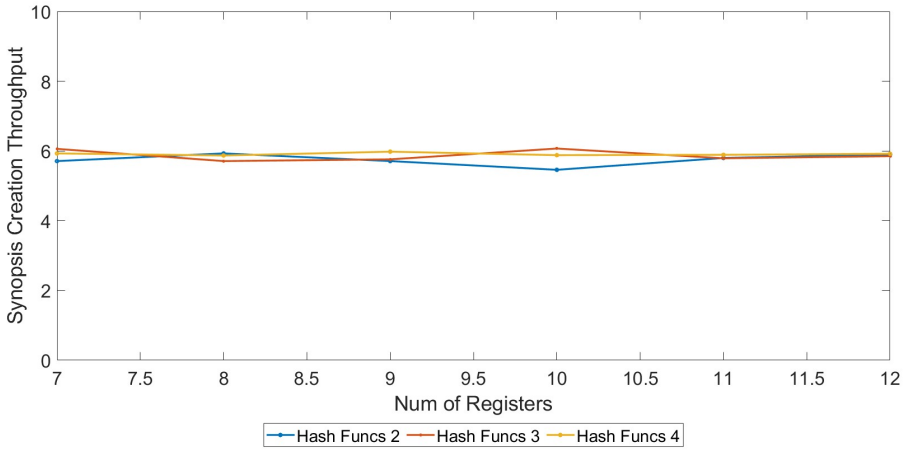
Synopsis Creation

The process of synopsis creation is an integral part of handling streaming data. Efficient synopsis creation ensures that the system can keep up with the incoming data stream. In evaluating the performance of a method, a crucial metric to consider is the time required to compare the query synopsis to the synopses created by the different data streams and provide an answer regarding the most similar one. This can be measured directly or represented as throughput, calculated using the formula:

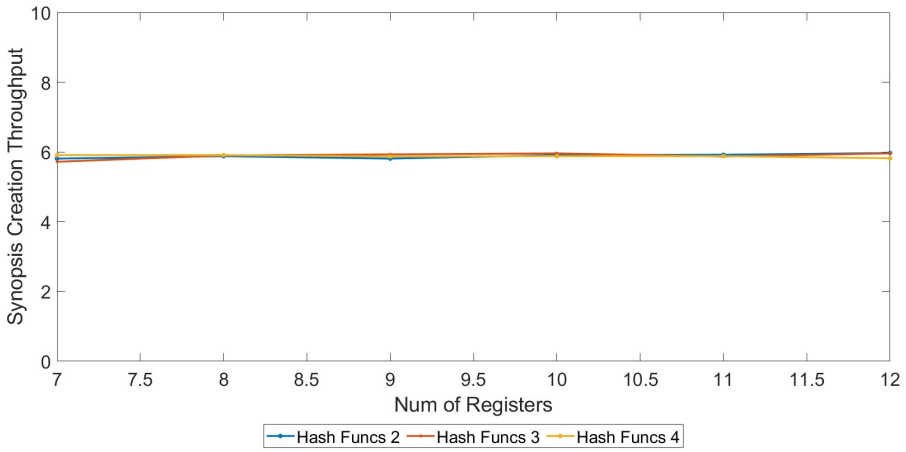
$$Throughput = \frac{\#ofOperations}{Time}$$

The first method that will be investigated is the one based on HyperLogLog. Users have the flexibility to modify two primary settings: the count of registers and the number of hash functions that will be used. As detailed in Condor's documentation, the register count can range from 4 to 16, with the value indicating the logarithmic scale of the actual register count. For instance, setting this value to 10, translates to the creation of a sketch that contains a total of 2^{10} registers. The number of hash functions refers to the number of replications that will be created and then averaged in order to get more accurate results. A series of tests were carried out to evaluate how the method reacts to various configurations. The outcomes are depicted in Figure 4.1 and Figure 4.2, where each line signifies a variant with a distinct count of hash functions. The x-axis denotes the register count utilized, while the y-axis displays the throughput of creating new synopses as it was recorded during the experiments.

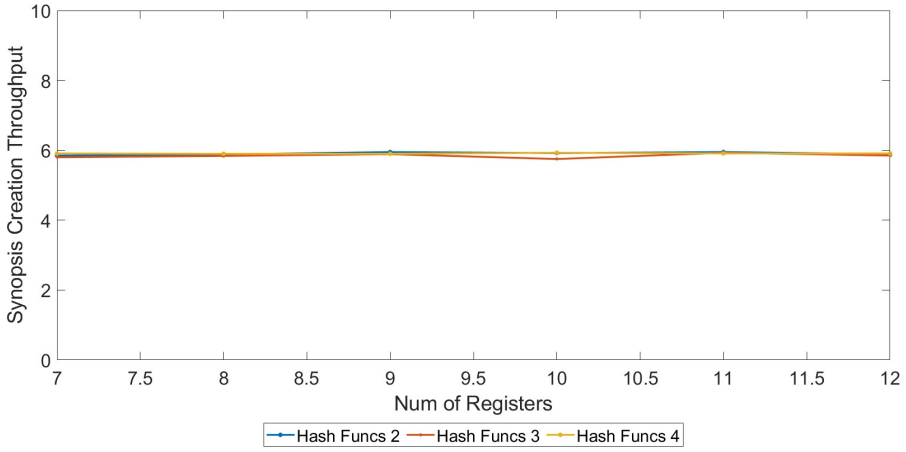
Upon reviewing Figure 4.1, it is apparent that for smaller window sizes, the average number of synopses generated each second remains consistent. Furthermore, the number of registers and the volume of incoming windows do not significantly



(A) N = 10000, W = 128



(B) N = 30000, W = 128



(C) N = 50000, W = 128

FIGURE 4.1: HLL synopses creation times in milliseconds for data streams with window size of 128 elements.

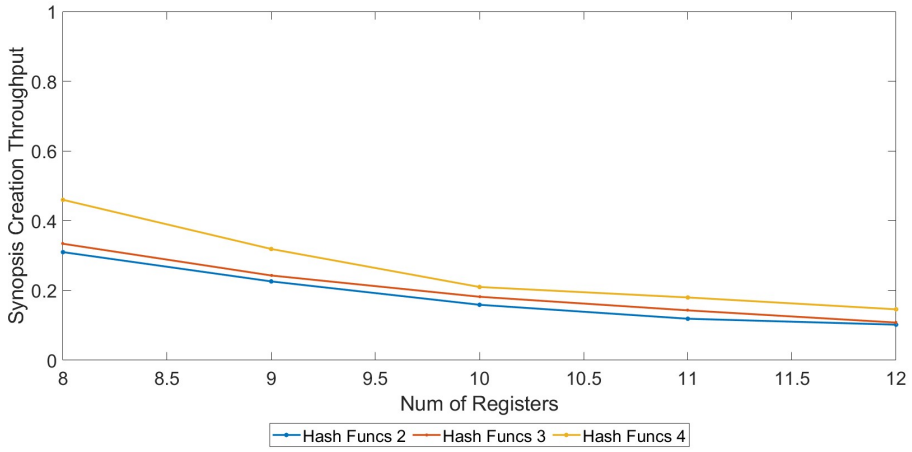
influence the throughput. The average throughput is constantly close to six synopses per millisecond which is the rate with which the data arrives from the Kafka pipeline to the system. This pattern changes when dealing with larger window sizes. When the window sizes increase, as shown in Figure 4.2, the rate of synopsis generation per second significantly drops. This is logical given that for a window size of 1024, the HyperLogLog algorithm is required to produce eight times the number of hashes compared to a window of size 128. Additionally, the larger window size necessitates a proportional increase in comparisons to ensure each hash value is allocated to the appropriate register.

By paying closer attention to the images it can be seen that, enhancing the register count consistently diminishes the throughput. Additionally, implementations with a greater count of hash functions exhibit lower throughput. This is consistent with previous discussions: an increment in the 'number of registers' parameter signifies a doubling of the actual register count since the actual count is 2 raised to the power of the given parameter. Also, increasing the 'number of hash functions' effectively means adding another complete set of registers, thus greatly escalating the method's complexity. Considering that each increment by one in the register count parameter effectively doubles the number of registers and, by extension, the computational workload, it is anticipated that the corresponding throughput plot would exhibit a steep, exponential decrease. This prediction aligns with the outcomes observed during the experimental process.

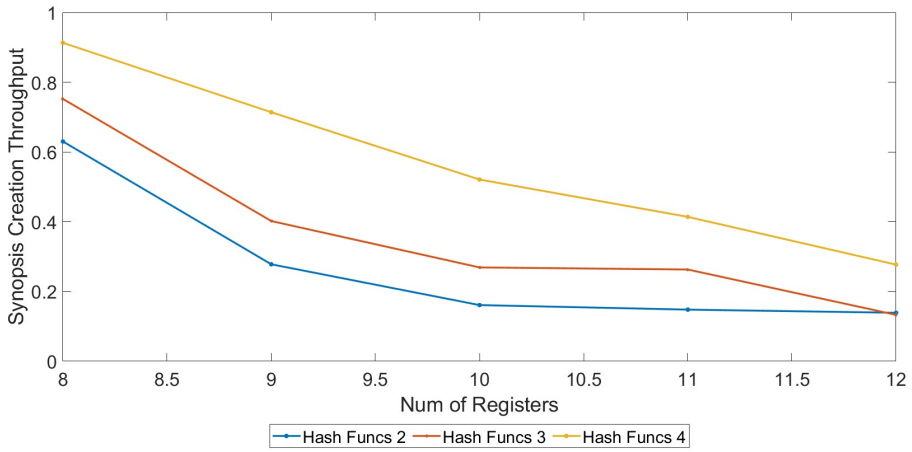
Moreover, when it comes to Theta sketches, it is a much more simple configuration the k value is the only parameter that can be adjusted. The value k, being the sole configurable parameter, determines the size of the sketch and consequently influences the throughput of synopsis creation. The mentioned figures aim to elucidate how different k values affect the number of synopses generated per millisecond, providing a visual comparative analysis to better understand the performance dynamics of theta sketches with varying configurations. The two figures show the system's behaviour for data streams of different window size. In the first figure the window size is 128 and in the second case each window consists of 1024 elements. Each sub figure shows how the system handles different number of windows. The system has to create 10, 30 and 50 thousand synopses respectively.

The observations reveal that the throughput remains stable with varying k values and the number of windows when the window size is 128. However, the significant factor influencing the number of synopses created per second is the window size. With a window size of 128, the system generates nearly 5 synopses per millisecond, but this throughput diminishes to 2 as the window size expands to 1024. Additionally, a slight decrease in throughput with an increasing k value is noticeable in the case of a 1024-element window. The rationale behind this can be traced back to the limited variations the k value can exhibit with a smaller 128-element window, as opposed to a larger 1024-element window where the alterations in k value can be more pronounced, thereby reflecting changes in throughput.

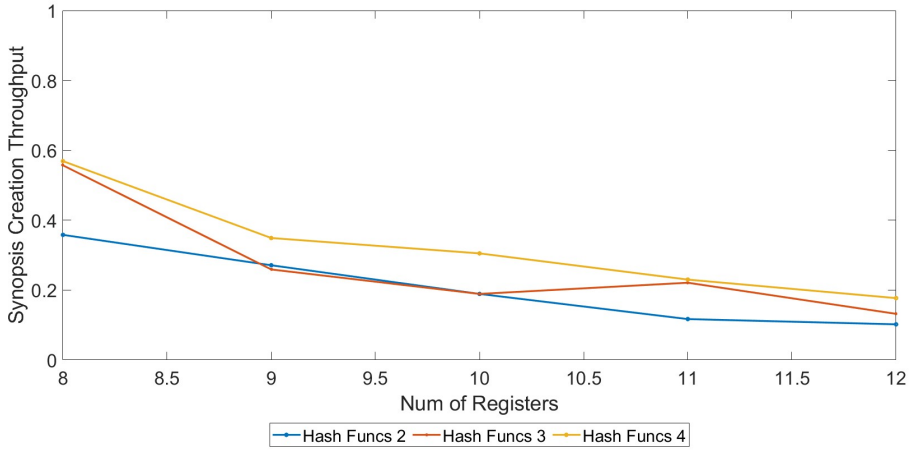
During experimentation carried out to understand the performance of DSTree tweaking the following configurable parameters was necessary: the amount of levels the synopsis will have, referenced also as tree depth, and node capacity. Through Figure 4.5 and Figure 4.6, the text illuminates how these parameters influence the rate at which synopses are generated per millisecond. This study is aimed at delving deeper into the workings of DSTree to potentially optimize its configuration for enhanced performance. By altering the tree depth and node capacity, the investigation seeks to elucidate the trade-offs and benefits associated with different configurations. The figures serve as a visual representation of these experiments, offering



(A) N = 10000, W = 1024

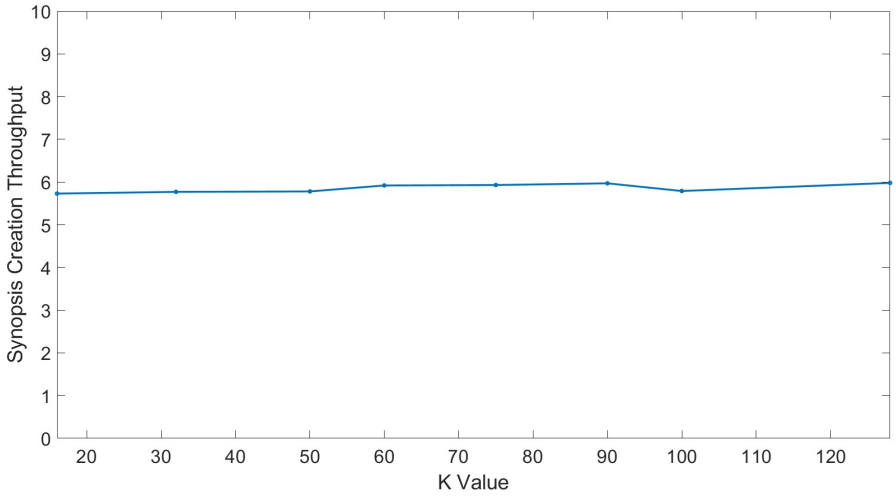


(B) N = 30000, W = 1024

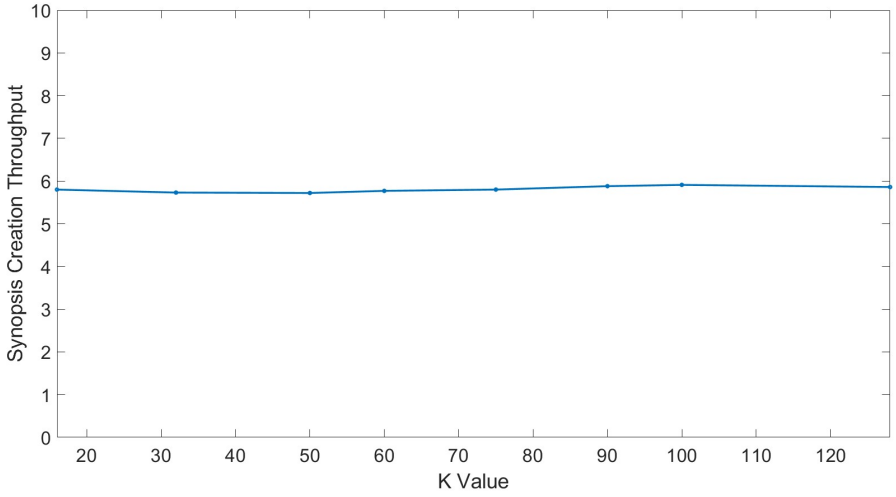


(C) N = 50000, W = 1024

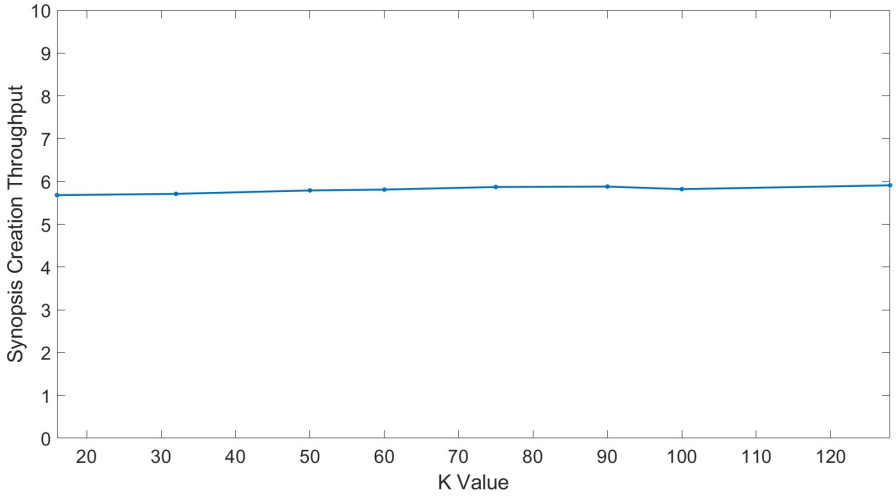
FIGURE 4.2: HLL synopses creation times in milliseconds for data streams with window size of 1024 elements.



(A) N = 10000, W = 128

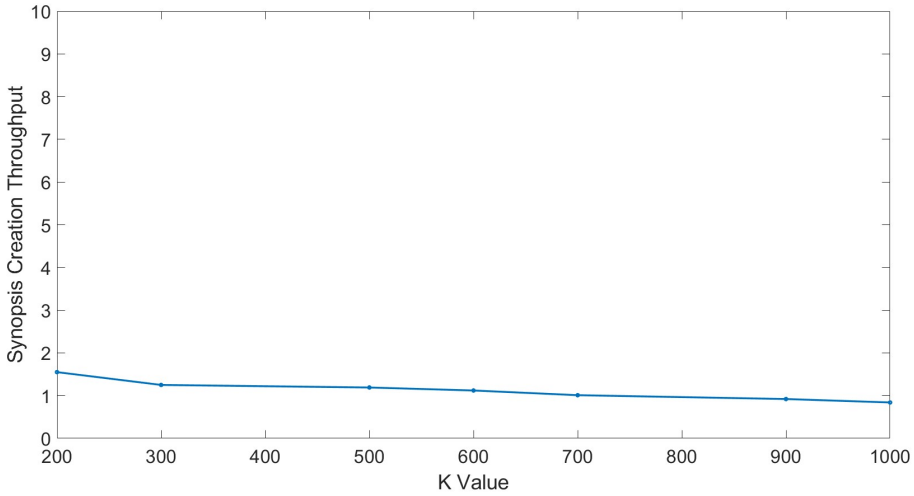


(B) N = 30000, W = 128

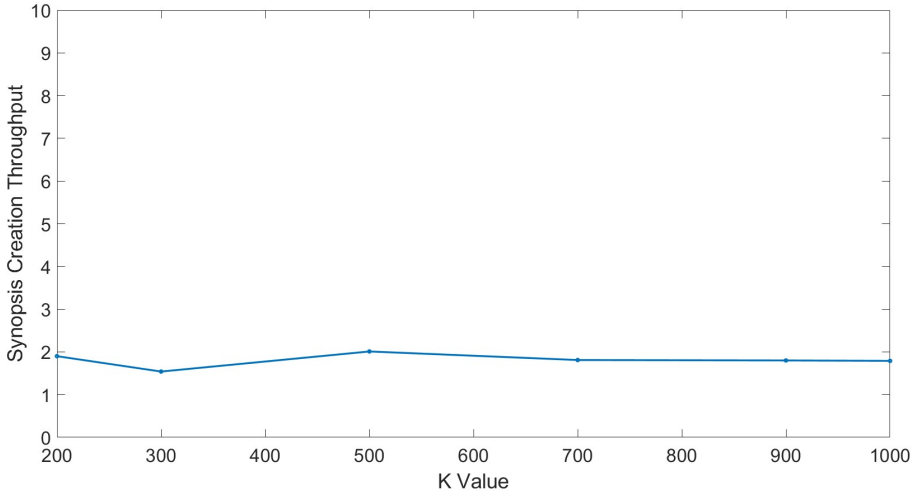


(C) N = 50000, W = 128

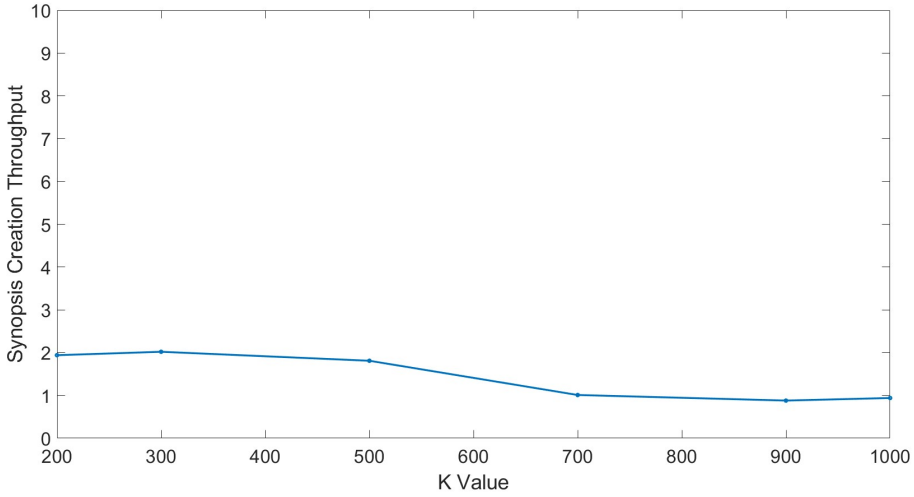
FIGURE 4.3: Theta synopses creation times in milliseconds for data streams with window size of 128 elements.



(A) N = 10000, W = 1024



(B) N = 30000, W = 1024



(C) N = 50000, W = 1024

FIGURE 4.4: Theta synopses creation times in milliseconds for data streams with window size 1024 elements.

insights into the relationship between the configurable parameters and the throughput of synopsis creation. Through this examination, a more nuanced understanding of DSTree's performance and the impact of its parameters is achieved, which could be instrumental in fine-tuning the method for specific use cases or computational environments.

The insights gathered from Figure 4.5 and Figure 4.6 indicate a consistent throughput, unaffected by varying configurations. Additionally, the window size doesn't appear to significantly impact the results. A minor decrease in throughput is observed with an uptick in node capacity, becoming more noticeable with a higher volume of data being ingested, especially when the tree depth is larger. This is logical, as a MetaData sketch synopsis with more layers offers a broader range of window segmentation possibilities, and hence, more potential node split scenarios to explore once a node reaches its capacity limit. While the additional time required for this is minimal, it accumulates with an increasing influx of data, leading to a rise in the average time for synopsis creation.

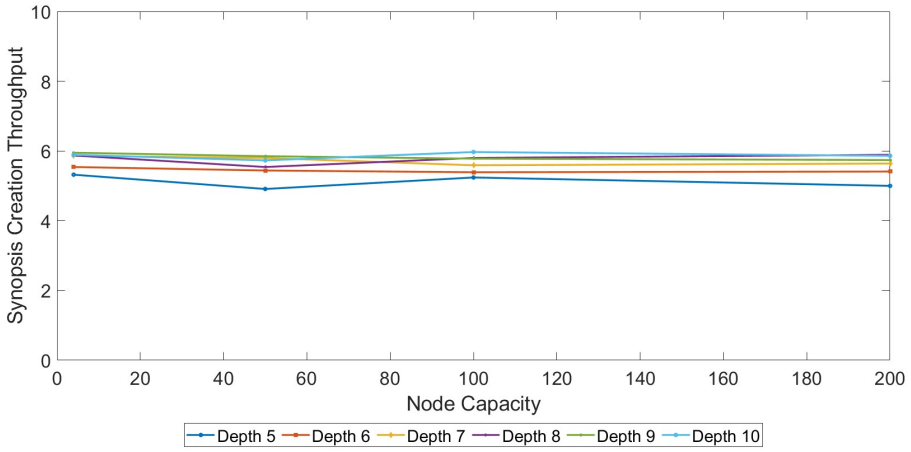
Similarity Calculation

Another crucial metric to consider is the time required to compare the query synopsis to the synopses created by the different data streams and provide an answer regarding the most similar one. The core functionality of these methods is to identify the most similar synopsis when given a query synopsis. The time taken to provide an answer is a direct measure of the efficiency of the method in handling query requests. This metric is vital as it impacts the latency of the system in responding to queries, which is a crucial aspect, especially in time-sensitive applications.

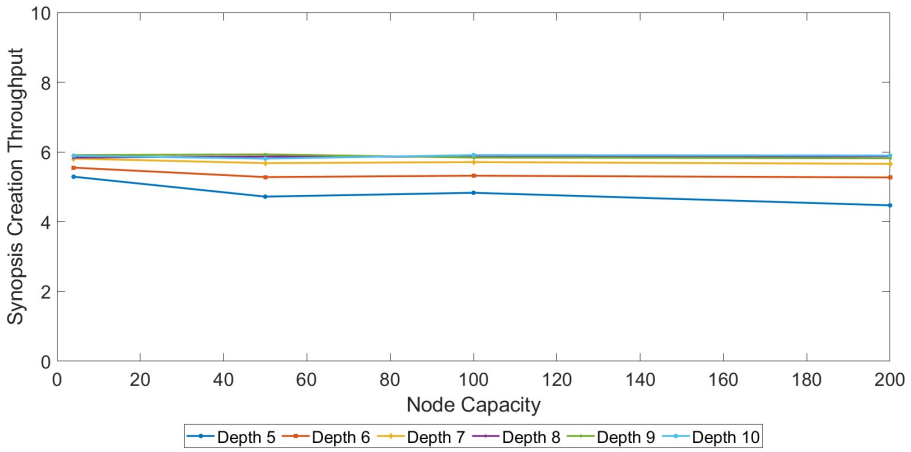
Figures 4.7 and 4.8 illustrate the throughput of the HyperLogLog (HLL) method when it comes to comparing a query with the already created sketches. These experiments, previously discussed, now focus on the system's capacity to handle similarity queries per second, and how many responses it can deliver within each scenario.

Moving on to the implementation based on Theta Sketches. The depicted outcomes in Figure x and Figure y show the influence of the k value on the quantity of answers the system can provide per second. It's clear that there's a decline in throughput as k increases. This pattern is explained when looking into the similarity calculation process. The approach involves contrasting the amount of mutual elements in two collections - the k smallest hashed values from the query stream and the k smallest hashed values from the data stream, holding onto the k smallest post-comparison. It's reasonable to deduce that as the size of these collections enlarge with a higher k, the time required for executing the comparisons and extracting the k smallest values also stretches. Hence, the throughput, indicative of the rate of answer production, shrinks as the comparison process becomes more time-intensive with an increasing k value.

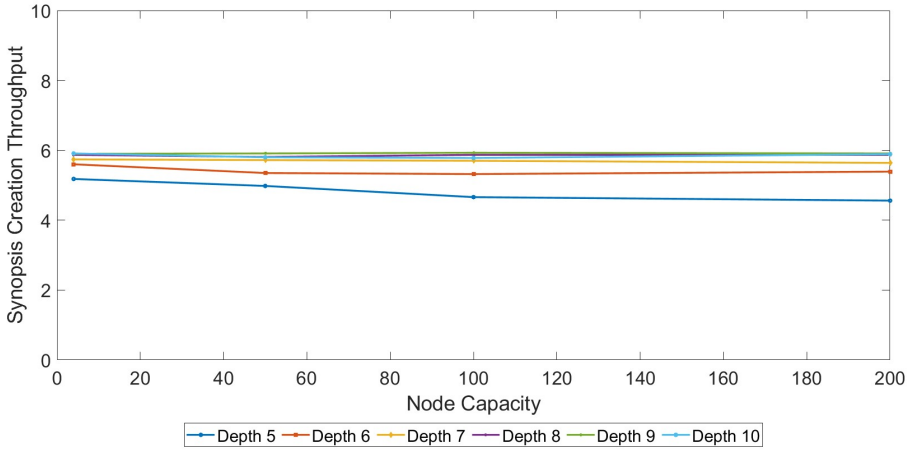
Figure 4.11 and Figure 4.12 illustrate the throughput of the DSTree method, showcasing the impact of varying its parameters. DSTree has multiple adjustable parameters, and our examination aimed at understanding how modifying each would influence the throughput. Among these parameters, the number of levels in the synopsis is of paramount importance as it's correlated to a more precise representation of the initial values, hence, enhancing precision as discussed in Chapter 3. In the figures, each line represents a different implementation of the method with varying numbers of levels. The diagrams display node capacity on the x-axis and throughput on the y-axis, where throughput is the key variable under investigation. The variety in node capacity further provides insight into how this variable influences



(A) N = 10000, W = 128

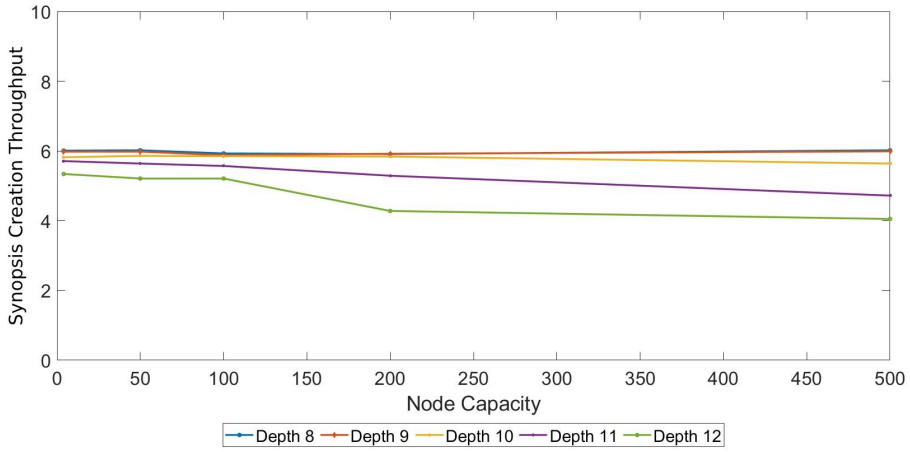


(B) N = 30000, W = 128

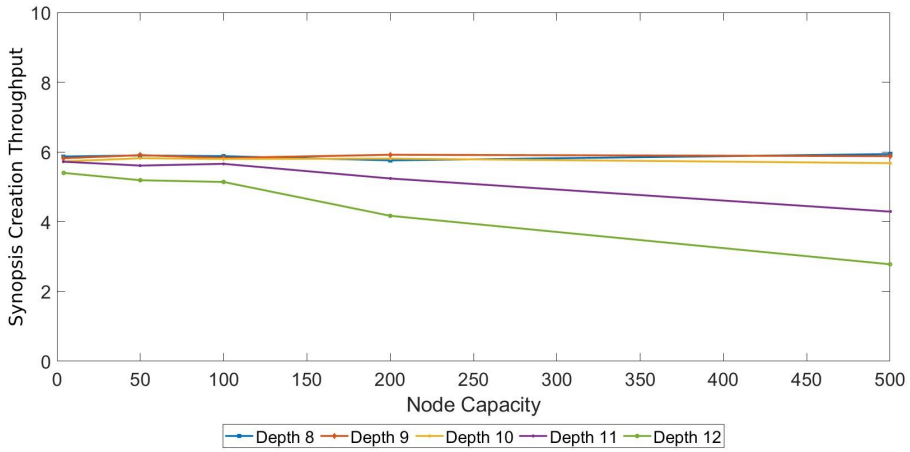


(C) N = 50000, W = 128

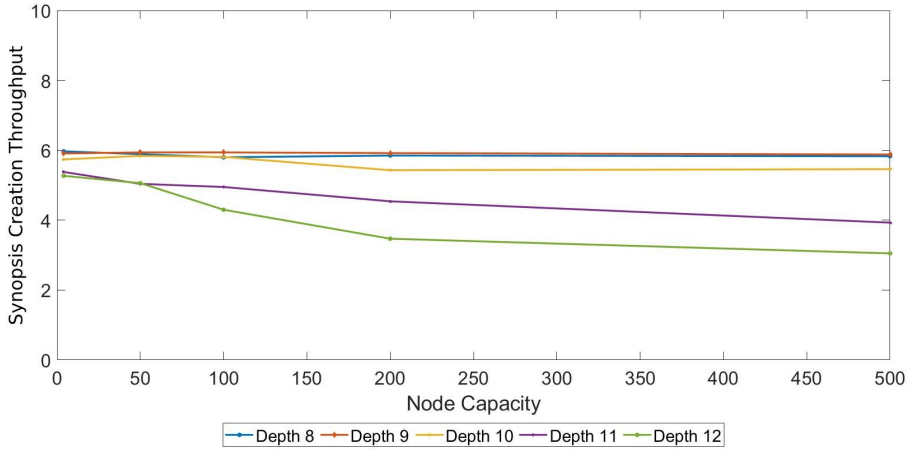
FIGURE 4.5: DSTree synopsis creation times in milliseconds for data streams with window size 128.



(A) N = 10000, W = 1024

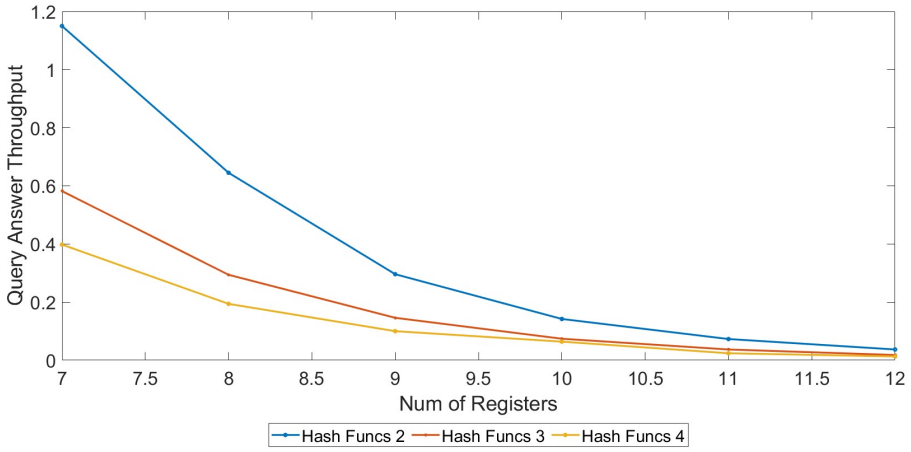


(B) N = 30000, W = 1024

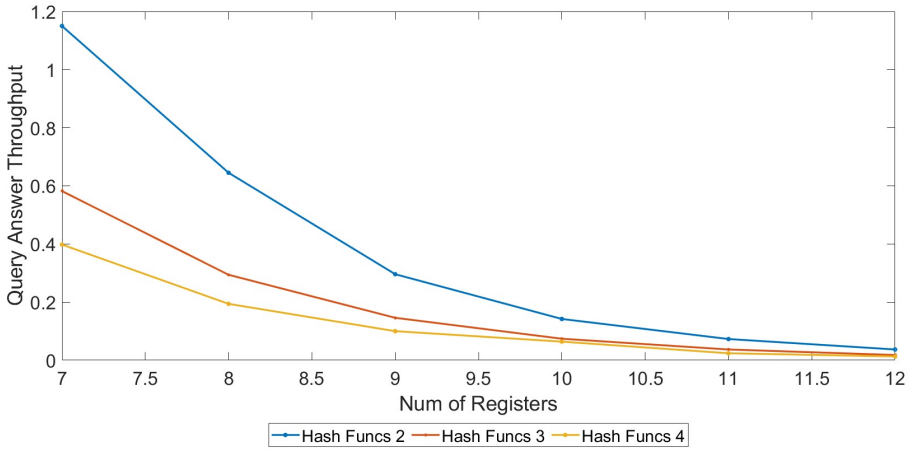


(C) N = 50000, W = 1024

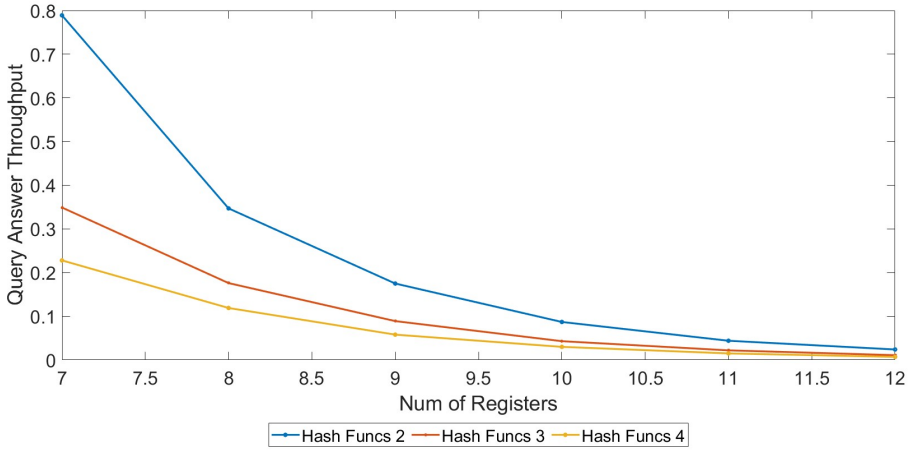
FIGURE 4.6: DSTree synopsis creation times in milliseconds for data streams with window size 1024.



(A) N = 10000, W = 128

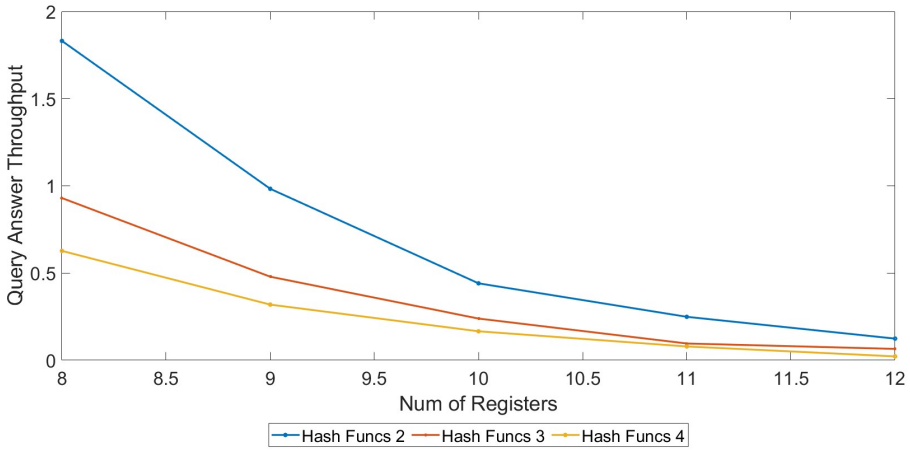


(B) N = 30000, W = 128

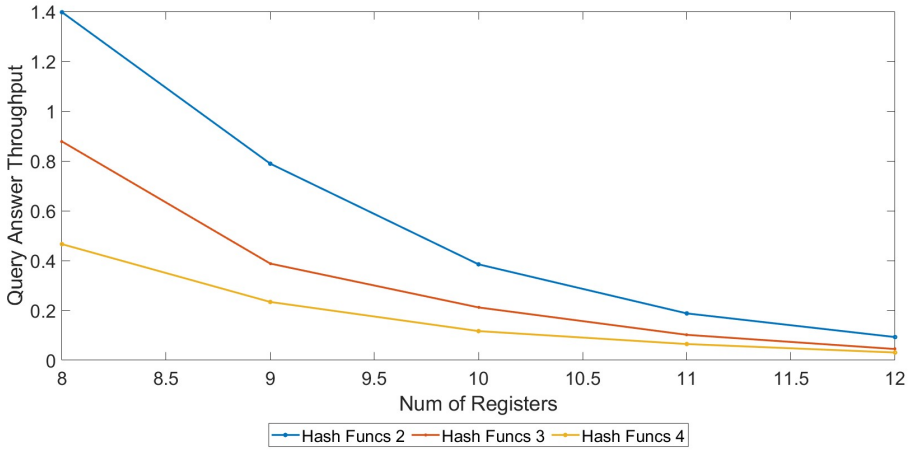


(C) N = 50000, W = 128

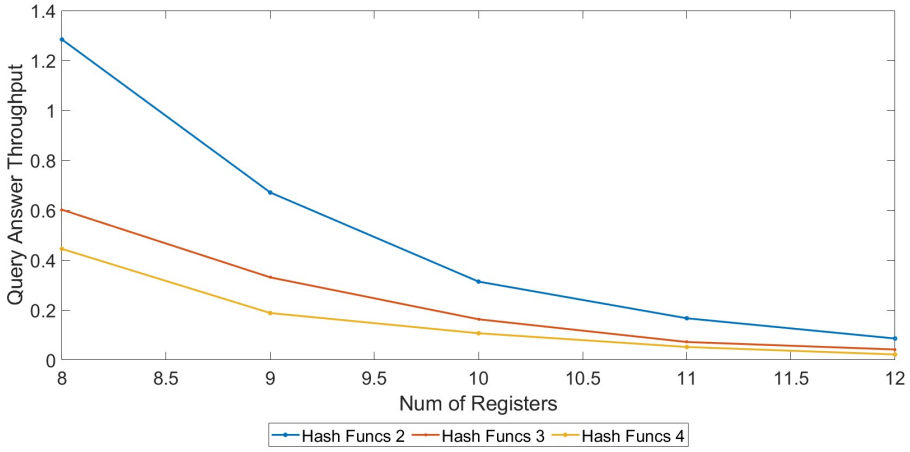
FIGURE 4.7: Number of queries the HLL based method can answer per millisecond, for window size equal to 128 and N number of synopsis.



(A) N = 10000, W = 1024

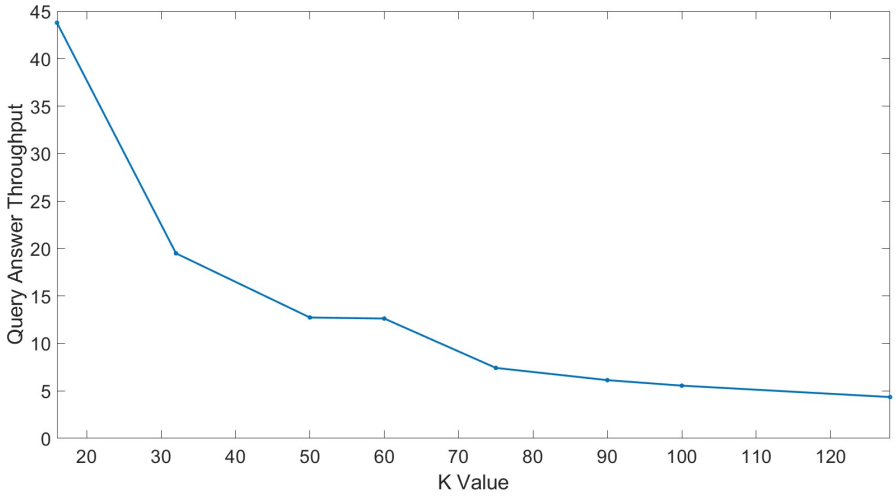


(B) N = 30000, W = 1024

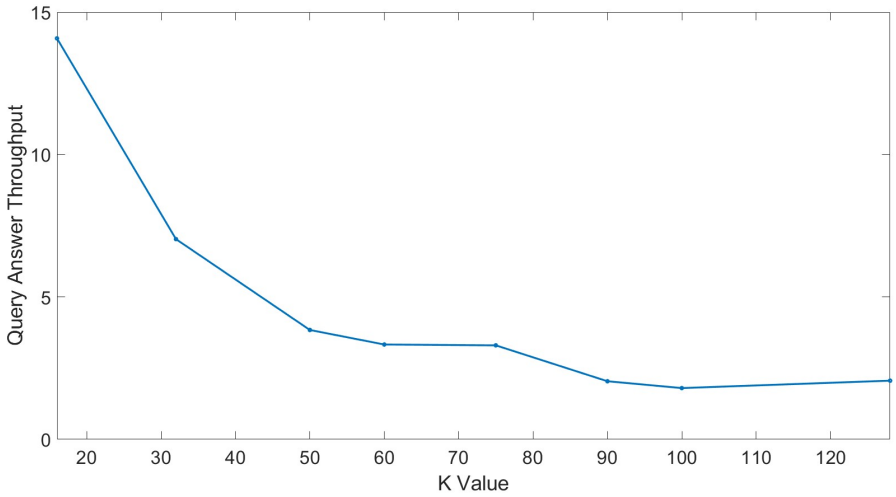


(C) N = 50000, W = 1024

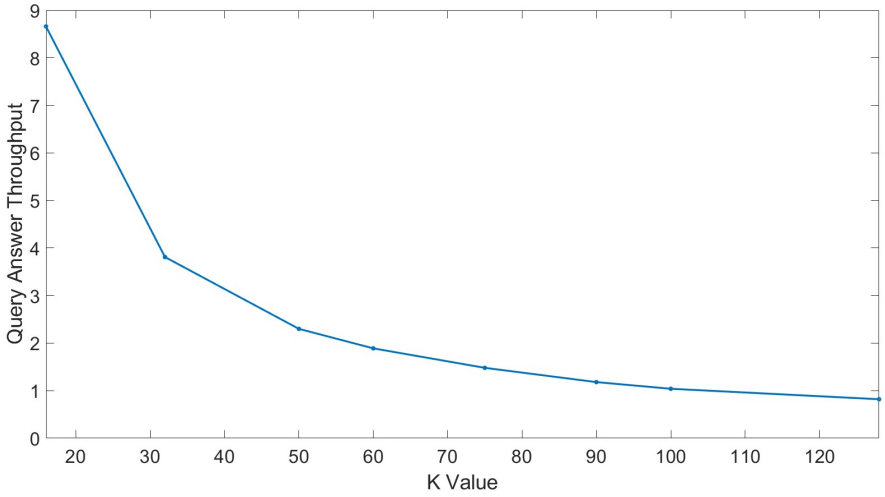
FIGURE 4.8: Number of queries the HLL based method can answer per millisecond, for window size equal to 1024 and N number of synopsis.



(A) N = 10000, W = 128

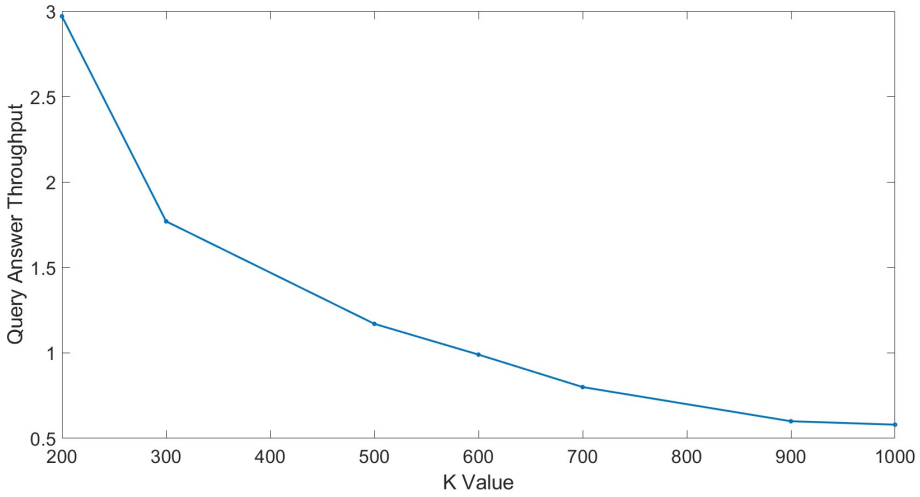


(B) N = 30000, W = 128

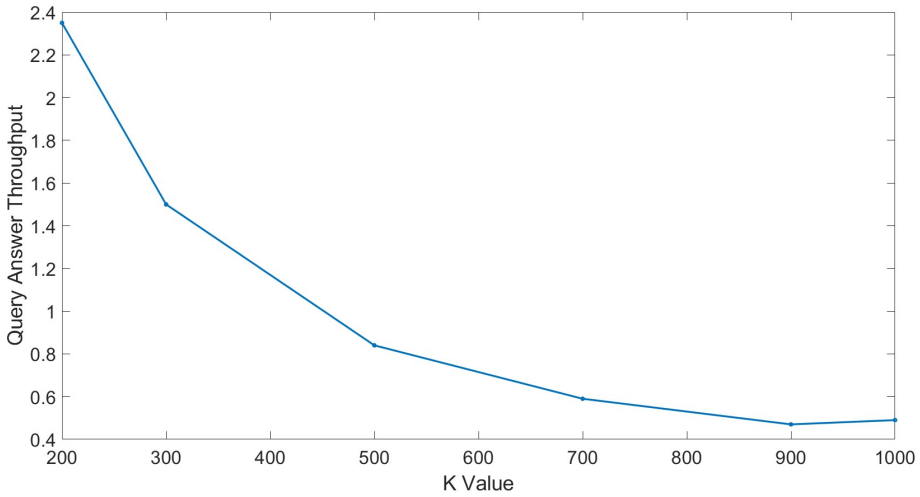


(C) N = 50000, W = 128

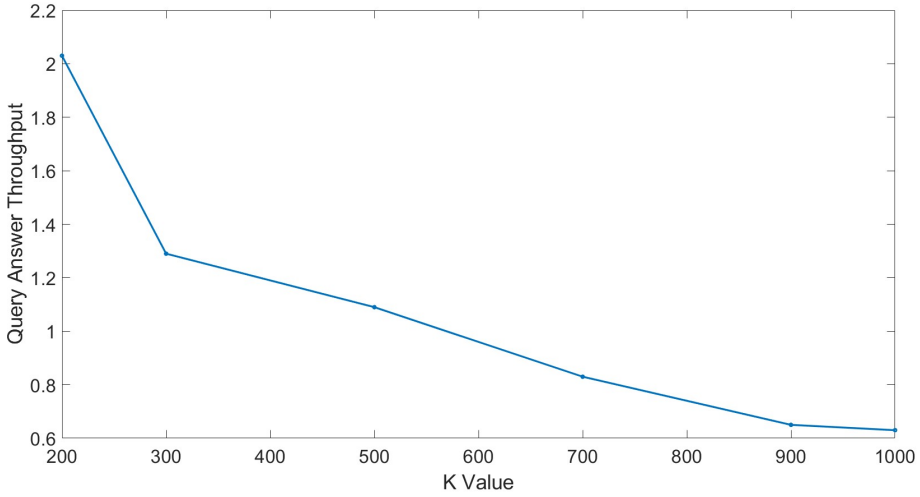
FIGURE 4.9: Number of queries the Theta based method can answer per millisecond, for window size equal to 128 and N number of synopsis.



(A) N = 10000, W = 1024



(B) N = 30000, W = 1024



(C) N = 50000, W = 1024

FIGURE 4.10: Number of queries the Theta based method can answer per millisecond, for window size equal to 1024 and N number of synopsis.

the throughput, rendering a comprehensive understanding of the DSTree method's performance across different configurations.

The analysis depicted in the figures reveals a relationship between the tree depth, node capacity, and throughput in DSTree. As anticipated, a rise in tree depth culminates in a reduction in throughput. This phenomenon can be attributed to the fact that the initial window is separated to more segments for each a mean value and a standard deviation has to be calculated. As the depth of the DSTree increases, so does the complexity of the recursive functions used to compute similarities between synopses, thus making the process becomes more computationally heavy and time-consuming.

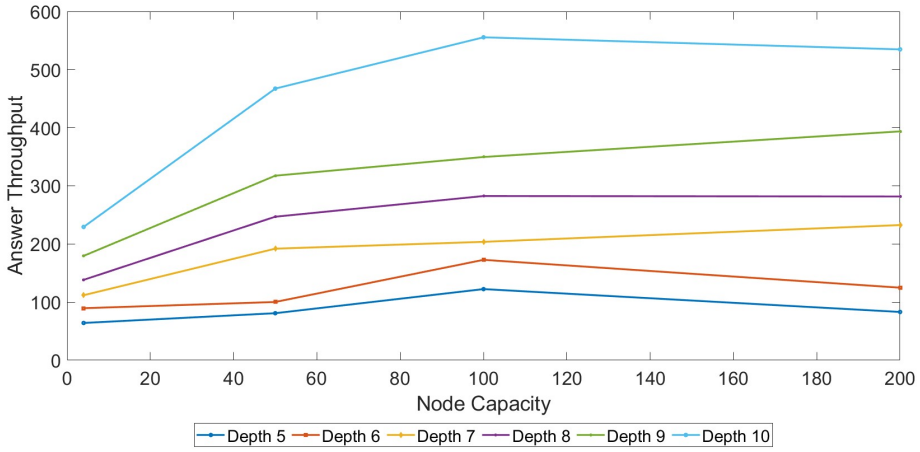
Contrastingly, a positive correlation is observed between the node capacity and throughput: an augmentation in node capacity leads to a boost in throughput. This is indicative of the system's enhanced ability to handle data as the node capacity expands. However, this seems to happen in an almost logarithmic rate and a point of saturation appears to be reached post a node capacity of 100, beyond which the throughput alterations become negligible. This plateau suggests that while augmenting node capacity can foster better throughput initially, there's a threshold beyond which the benefits taper off. Hence, while configuring the DSTree, a judicious selection of node capacity and tree depth is imperative to balance computational efficiency against the precision of data representation, ensuring the system is tailored to the demands of the specific use case it is deployed for.

Another dimension of the DSTree that warranted investigation was its adaptability to windows of varying sizes. The differing window sizes present distinct challenges and understanding how the DSTree copes with these variations is essential for evaluating its robustness and flexibility. In the experiments illustrated in Figure 4.11, the window size is set at 128, while in Figure 4.12, it is expanded to 1024.

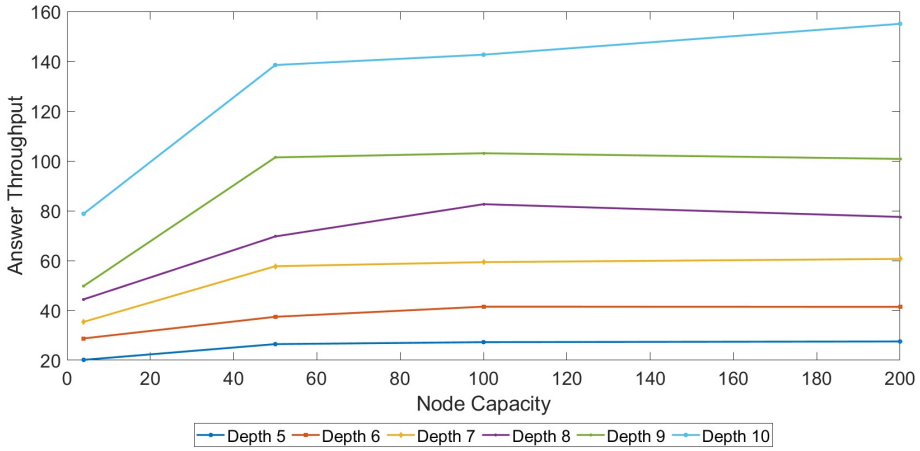
A notable observation from the diagrams is that a DSTree with a depth of 10 can generate over 100 answers per second, with a window size of 128. However, this capability diminishes when the window size is increased to 1024, with the number of synopses created dropping to 60 per second. However, after repeating the experiment for other window sizes we realized that the throughput did not increase linearly. A more complex behavior was observed. This variation in performance based on window size is more vividly depicted in Figure 4.13, which provides a clearer comparative view.

The rationale for this observed behavior is as outlined below. A tree depth of 10 signifies that the lower level of the arrays that hold the statistical information, as seen in Chapter 3.3.3, will accommodate a maximum of 512 values. If the window size considerably exceeds this, the last level will store the average of the elements that will arrive in a given window. For instance, with a window size of 2048, when the synopsis will be created, each tuple in the lowest level in the synopsis arrays will retain the average of 4 consecutive incoming elements. This scenario diminishes the precision of calculations during the estimation of distance between two synopses. Consequently, when the function 'groupDistanceCalc' is invoked, a higher number of nodes will be pruned. This reduces the count of necessary calculations for assessing similarity, albeit at the cost of accuracy.

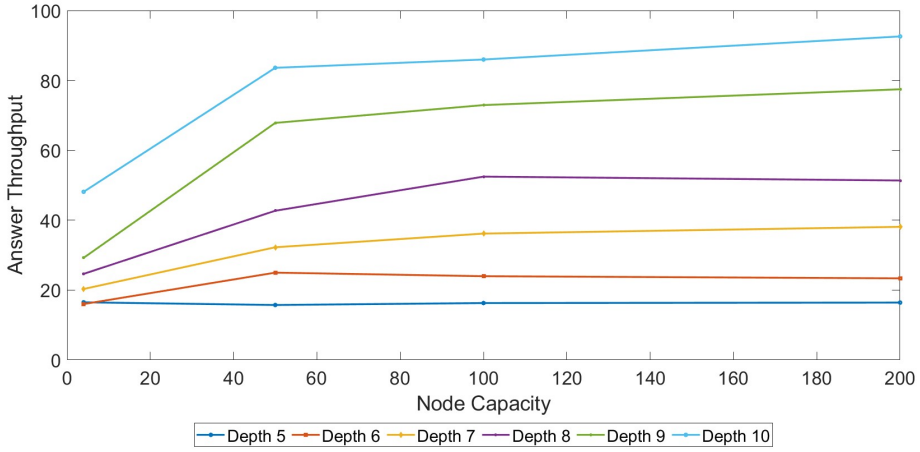
When the widow size is smaller, the number of calculations required to estimate similarity will also be smaller. Hence, more answers can be provided per second. So, that explains the higher throughput for window size 128.



(A) N = 10000, W = 128

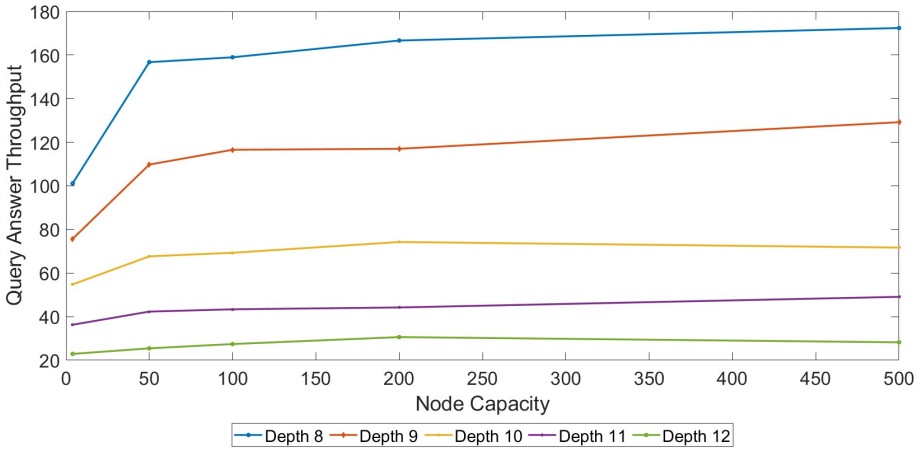


(B) N = 30000, W = 128

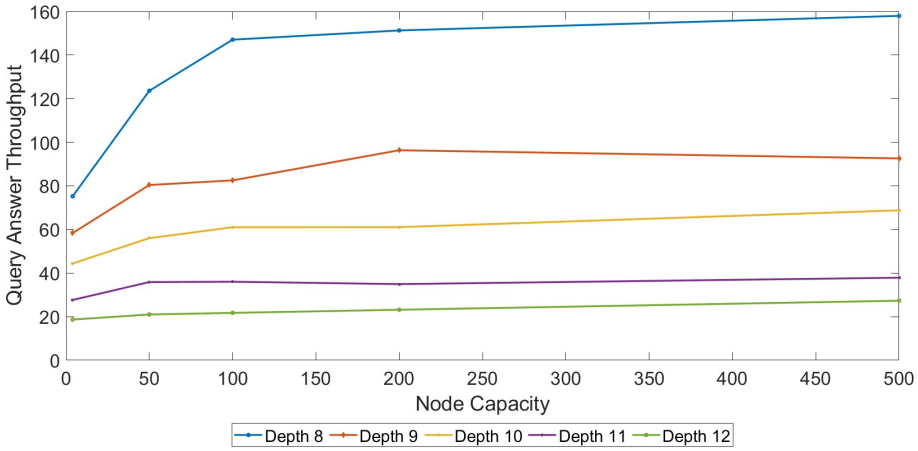


(C) N = 50000, W = 128

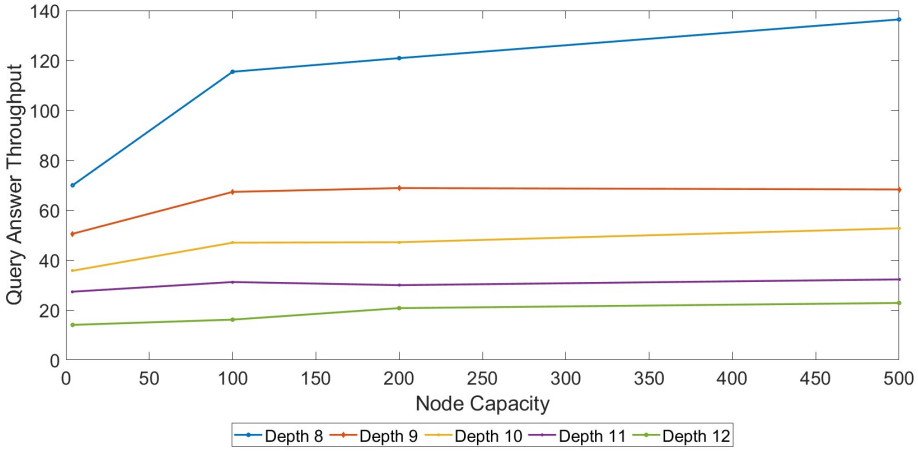
FIGURE 4.11: Number of queries the DSTree based method can answer per millisecond, for window size equal to 128 and N number of synopsis.



(A) N = 10000, W = 1024



(B) N = 30000, W = 1024



(C) N = 50000, W = 1024

FIGURE 4.12: Number of queries the DSTree based method can answer per millisecond, for window size equal to 1024 and N number of synopsis.

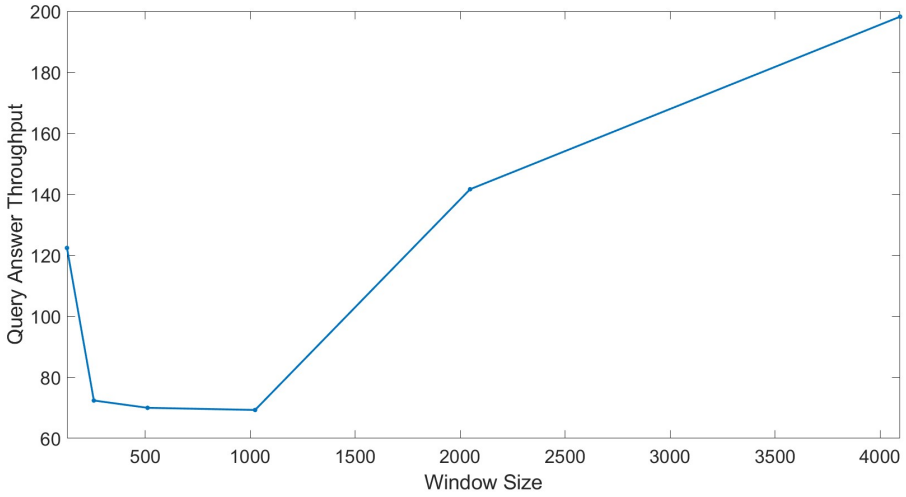


FIGURE 4.13: Answer Throughput for various window sizes.
Tree depth: 10. Node capacity: 100.

4.4 Accuracy Evaluation

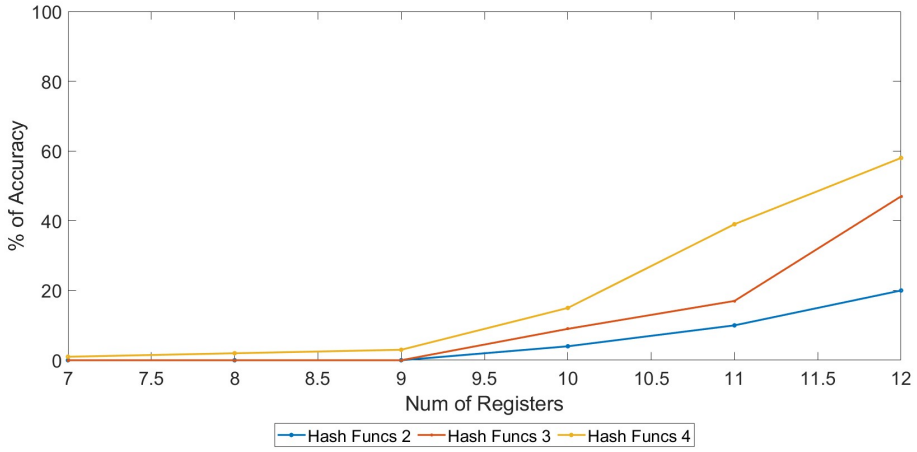
In our investigation, we delve into the precision of established data synopsis methods—DSTree, HyperLogLog (HLL), and Theta Sketches—across a spectrum of configurations. The evaluation is comprehensive, examining how tree depth, node capacity, register count, and hash functions influence each method's accuracy. We further scrutinize how these techniques perform under varying window sizes.

Additionally, in order to evaluate the accuracy of each method,the dataset by Laurent Amsaleg and Hervé Jégou will also be incorporated. It includes 10,000 windows of 128-size samples and 100 equivalent-sized queries with a reference file for the closest time series match. This dataset, alongside the custom datasets crafted from the generator we developed, and have been used in the previous chapter experiments, serves as the foundation for our accuracy assessments.

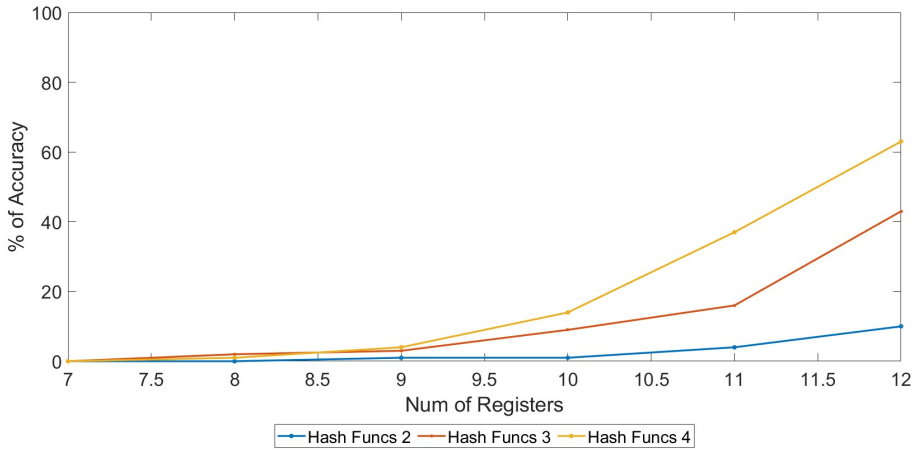
Firstly, the findings for the HyperLogLog (HLL) technique will be displayed, following the same format as the preceding chapter. Figure **??** depicts the performance variations with each curve corresponding to a distinct number of hash functions utilized. The x-axis indicates the number of registers involved, while the y-axis quantifies the method's accuracy, depicted as the percentage of accurate responses delivered.

Given the probabilistic nature of the synopses and the potential similarity of data from the random generator, a response is considered accurate if it ranks within the top ten most similar, rather than only the most similar. This approach is justified by the fact that, against the backdrop of 10,000 to 50,000 synopses, the top ten represents a minimal subset, thus still ensuring a high standard for accuracy.

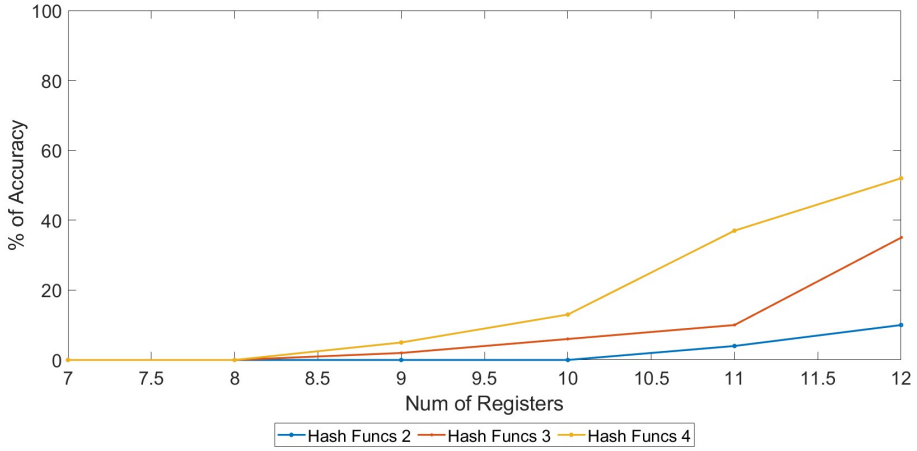
The data clearly indicates that in both cases, accuracy improves with a rise in the number of registers, showing a positive correlation between register count and hash function quantity on accuracy. Additionally, it's more efficient to boost accuracy by augmenting the number of hash functions rather than registers. Unlike increasing registers, which doubles memory and computation needs, adding more hash functions does not have as significant an impact, since it only introduces an additional set of registers without doubling the existing ones. Finally, the number of time series compared doesn't have an impact on the accuracy of the method.



(A) N = 10000, W = 128

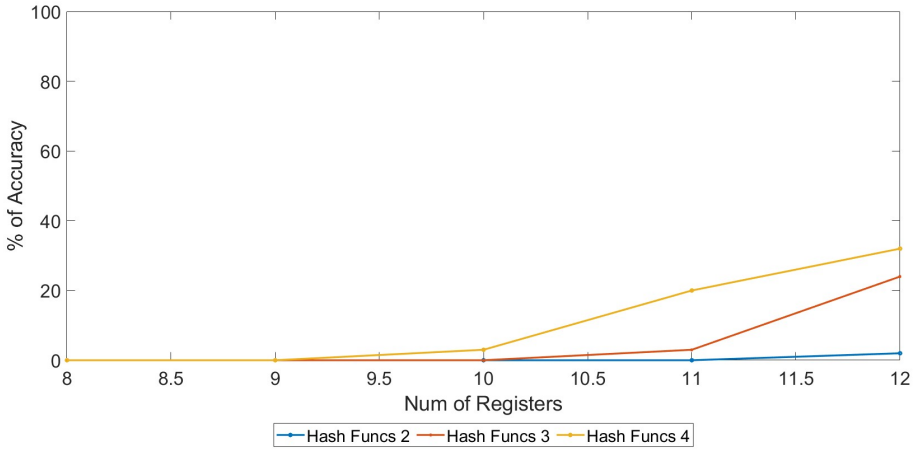


(B) N = 30000, W = 128

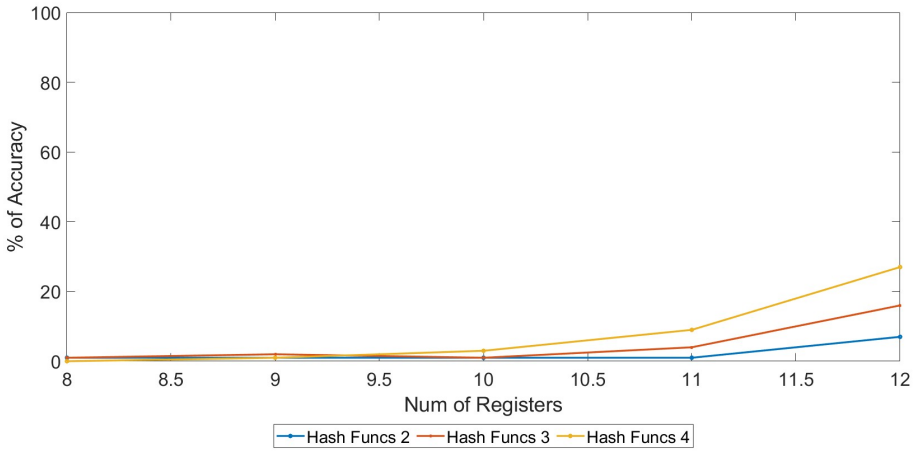


(C) N = 50000, W = 128

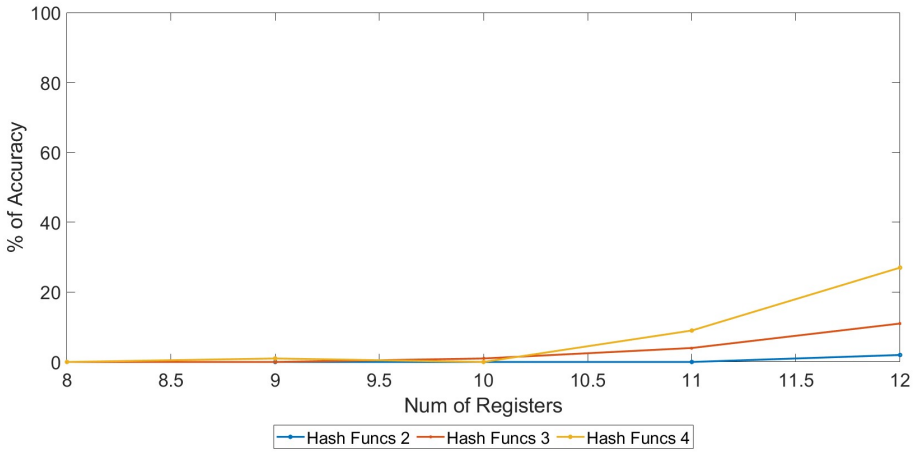
FIGURE 4.14: Accuracy of the HLL based method, for window size equal to 128 and N number of synopses.



(A) N = 10000, W = 1024



(B) N = 30000, W = 1024



(C) N = 50000, W = 1024

FIGURE 4.15: Accuracy of the HLL based method, for window size equal to 1024 and N number of synopses.

In Figure 4.14, the results from the dataset by Laurent Amsaleg and Hervé Jégou are presented, which comprises 10,000 distinct time series, each of size 128.

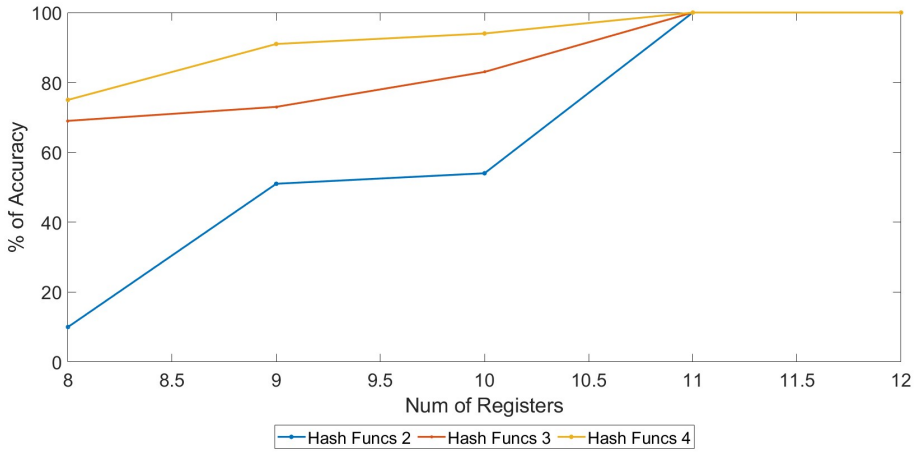


FIGURE 4.16: Accuracy results of the HLL based method and where the dataset created by Amsaleg and Jégou was used as input. N = 10000, W = 128

It is obvious that the method shows a great improvement in accuracy when handling data coming from the second dataset. This is more evident in Figure 4.17, where a comparative analysis can be conducted. The marked improvement in accuracy for this dataset, as compared to the custom-generated dataset, can be attributed to the distinct statistical characteristics of the two datasets. Crucially, as noted in [3], when the intersection size of two sets considerably exceeds the union size, the accuracy of similarity calculations using HLL-based methods tends to be lower.

Figure 4.17 shows the performance of the algorithm when having to deal with a dataset of 10.000 time series of size 128 coming from the custom made dataset, one of size 128 coming from the second dataset and one of size 1028 coming again from the custom data set. The number of hash function is set to four, but the results are similar when that number changes.

Another interesting observation is that the accuracy of the method reduces when the window size increases but the number of registers remains the same. This is easily explained since with the larger number of elements in a given window a larger synopsis has to be created in order to more efficiently describe them.

Following the previously established format when analyzing the accuracy of Theta Sketches, he outcomes for window sizes of 128 and 1024 are displayed in Figures 4.18 and 4.19 respectively. The analysis is relatively straightforward since there's only one parameter to consider. Also, in Figure 4.20 the results from the second dataset are presented.

The performance of the algorithm appears consistent across various conditions, with efficiency improving linearly as the parameter k increases. Notably, the algorithm's effectiveness is not influenced by larger window sizes or a higher influx of incoming time series.

As observed with the HyperLogLog (HLL) method, the dataset supplied by Amsaleg et al. exhibits distinct characteristics. There is an increase in accuracy, indicating that this particular dataset interacts with the Theta sketch method in a manner

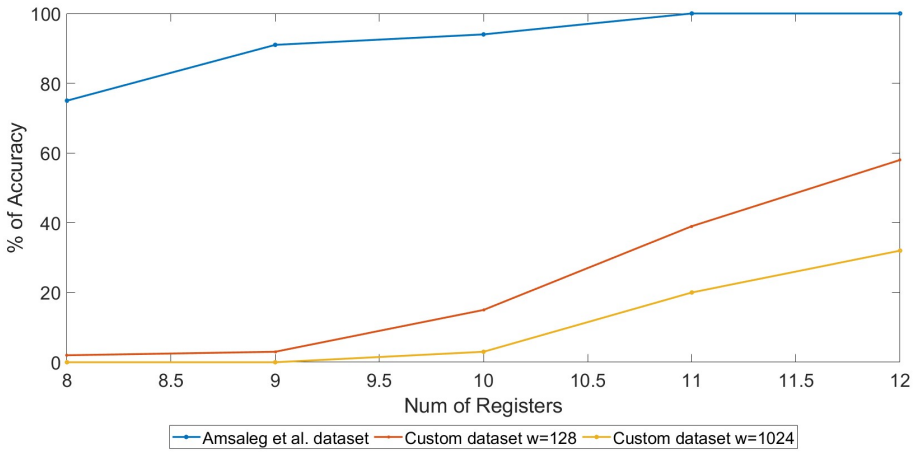


FIGURE 4.17: Accuracy results for the HLL based method for datasets of different sizes and statistical properties.

that enhances its ability to correctly estimate similarities between time series data. This could be attributed to the dataset's unique statistical properties.

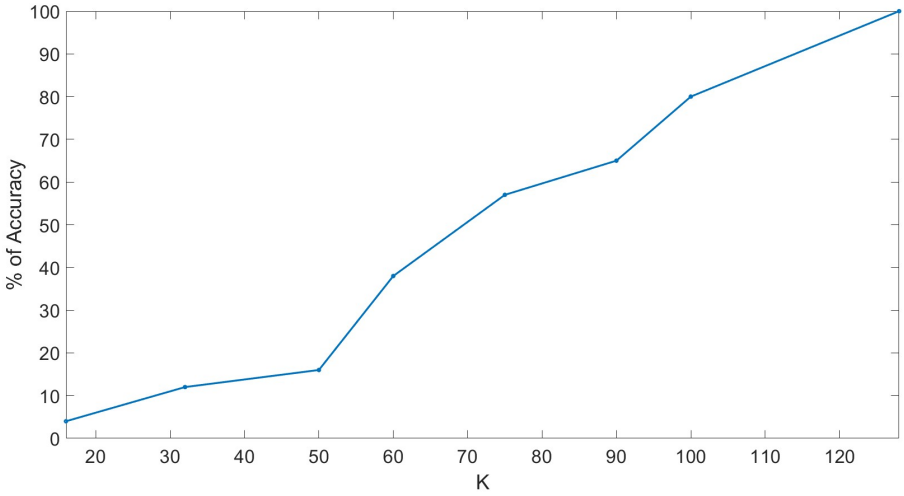
Last but not least, the final segment of the experimental results focuses on the streaming implementation of the DSTree method. Figures 4.21 and 4.22 detail how the accuracy of similarity estimation is affected by two parameters: the depth of the synopsis and the node capacity. Similar to previous methods, tests were also carried out using the second dataset provided by Amsaleg et al., with outcomes illustrated in figure 4.23. These figures collectively aim to shed light on the efficacy of DSTree's parameters in producing accurate similarity estimations in a streaming data context.

The analysis of the results reveals a key insight: the accuracy of the DSTree method is significantly enhanced by increasing the number of levels in the synopsis. This improvement is anticipated because more levels mean less aggregation is needed in constructing the synopsis, allowing for a finer representation of the original time series. Hence, the synopsis retains a higher fidelity to the original data, leading to more precise similarity estimations. This is also verified from the fact that for window size of 128 and depth greater than seven, the accuracy reaches one hundred percent, as in that case the synopsis is so big that holds every element from the original data.

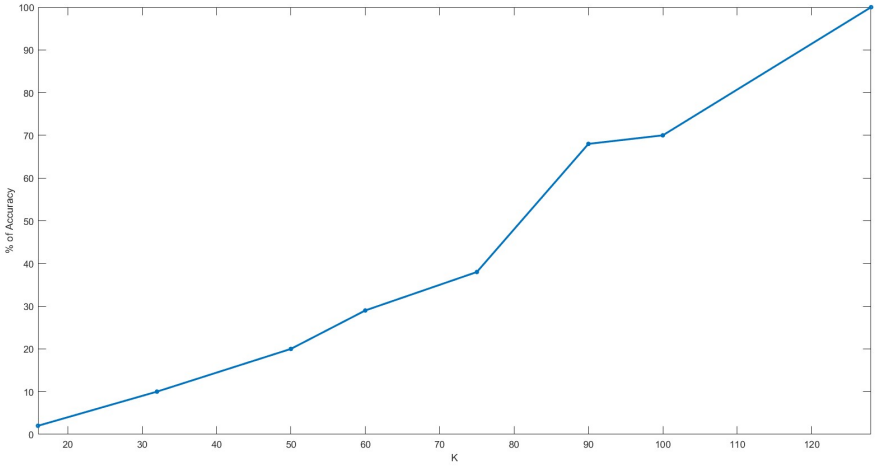
The observations also indicate that altering the node capacity within the DSTree method does not substantially affect accuracy. Therefore, when adjusting this parameter, it's crucial to consider its influence on throughput rather than on accuracy. This suggests that while node capacity may affect processing speed or memory usage, it does not necessarily contribute to the precision of the similarity estimations between time series data.

Another aspect impacting accuracy is the volume of data series integrated into the tree. The results demonstrate that the accuracy slightly diminishes when the input consists of 50,000 data series compared to when the tree comprises 30,000 or 10,000 synopses. This discrepancy is more pronounced for windows of larger sizes, suggesting that the tree's performance in accurately representing the data may degrade as the quantity of data increases, particularly when dealing with more extensive data windows.

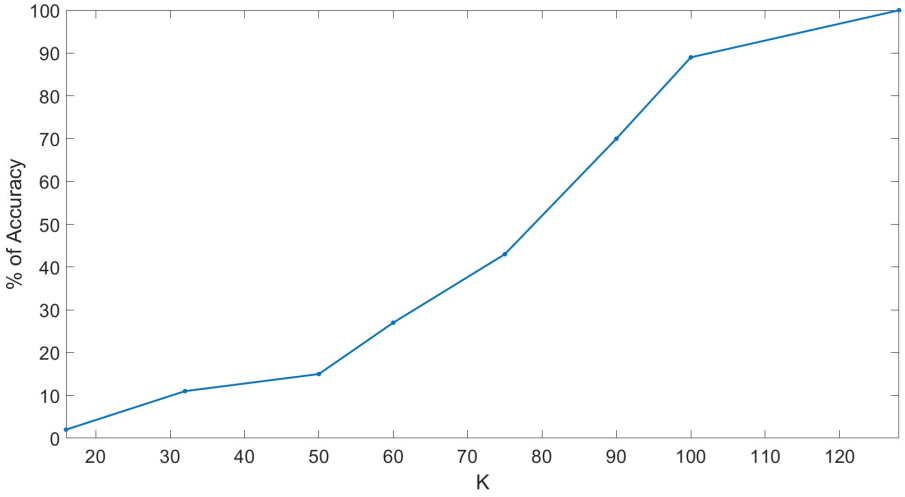
A plausible explanation for the decrease in accuracy as the tree size increases



(A) N = 10000, W = 128

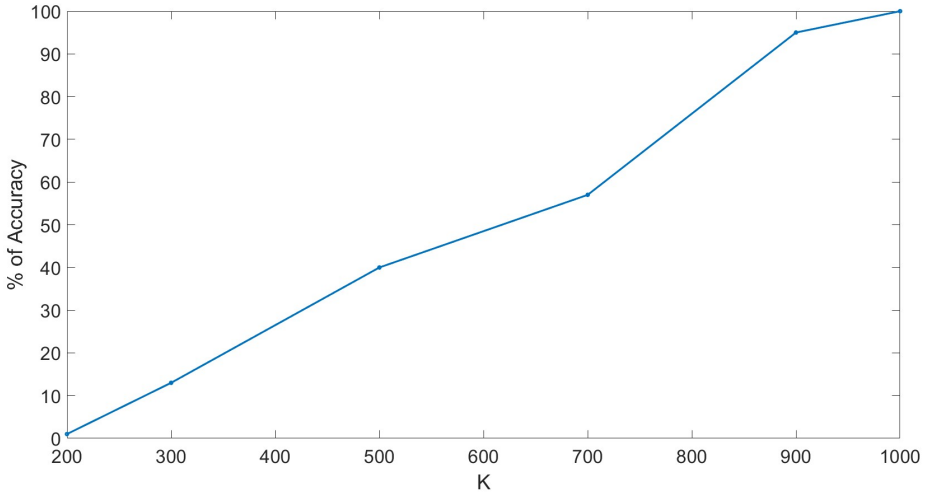


(B) N = 30000, W = 128

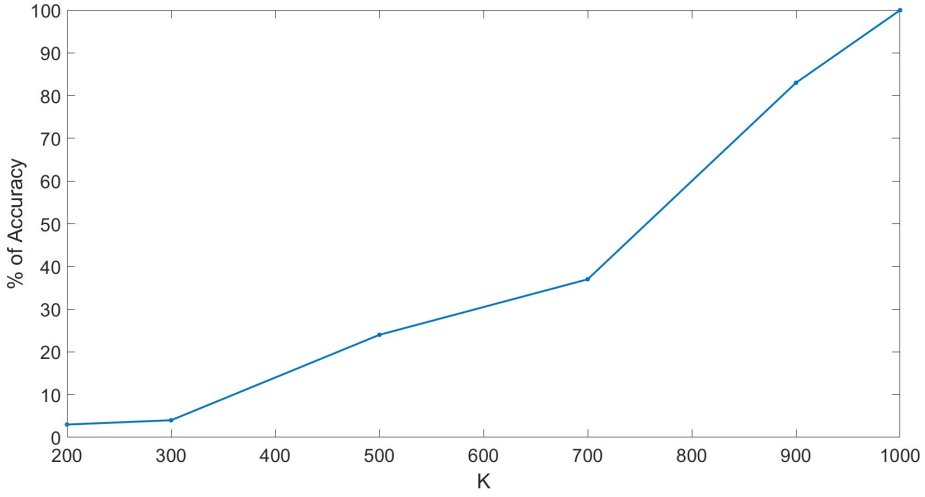


(C) N = 50000, W = 128

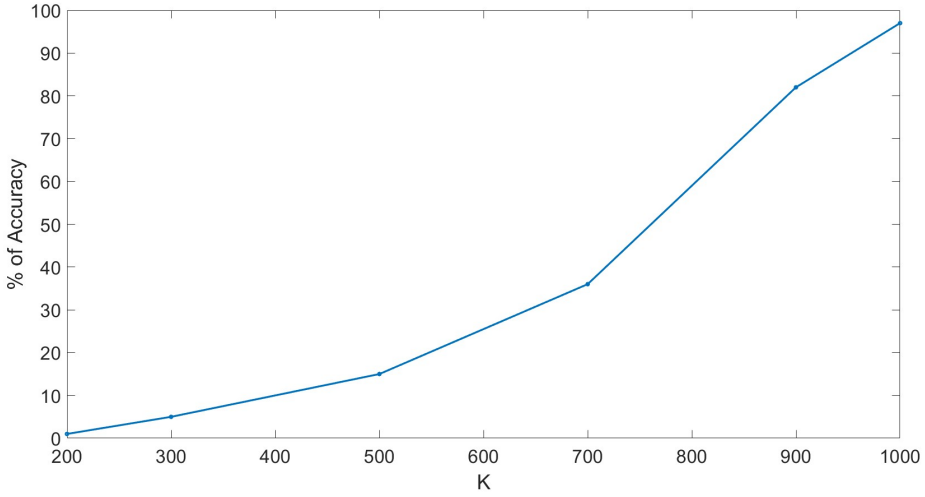
FIGURE 4.18: Accuracy of the Theta Sketch based method, for window size equal to 128 and N number of synopses.



(A) N = 10000, W = 1024



(B) N = 30000, W = 1024



(C) N = 50000, W = 1024

FIGURE 4.19: Accuracy of the HLL based method, for window size equal to 1024 and N number of synopses.

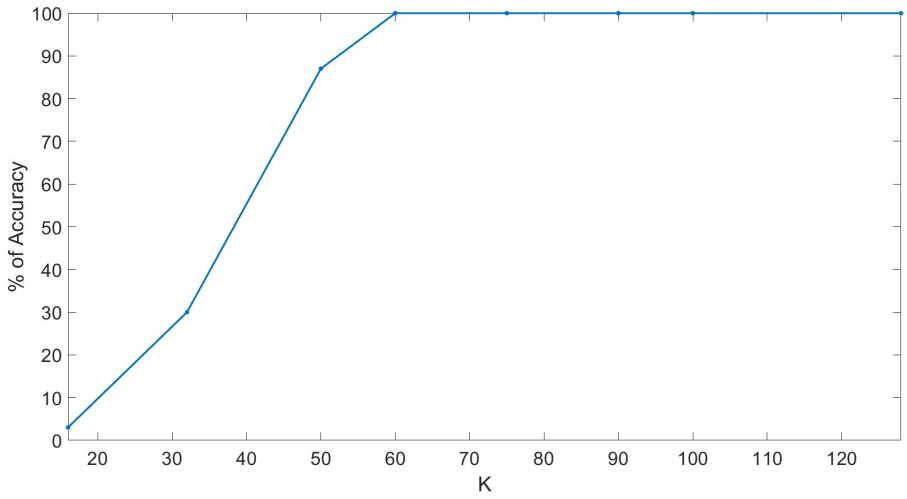
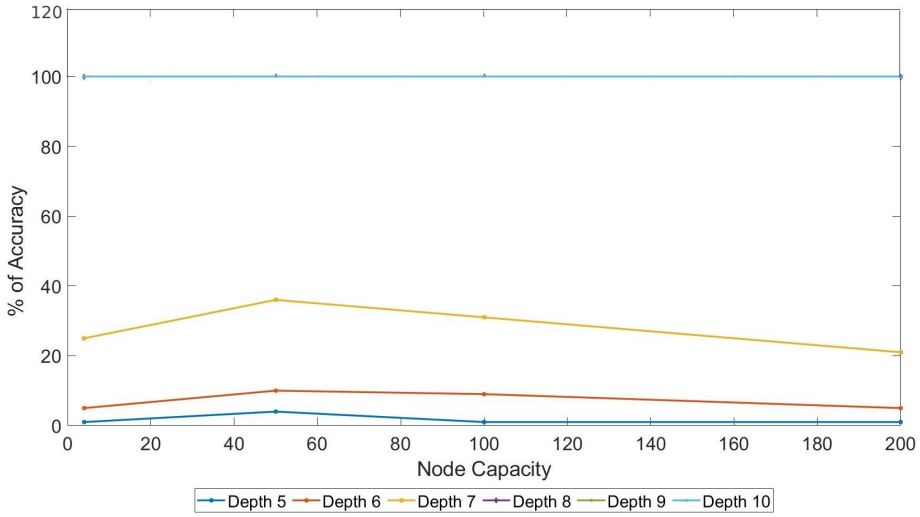


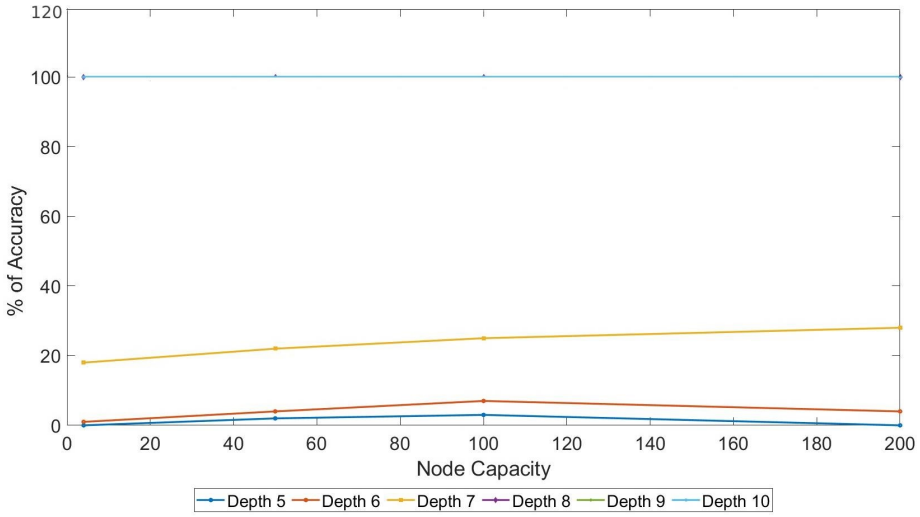
FIGURE 4.20: Accuracy results for the Theta Sketch based method and where the dataset created by Amsaleg and Jégou was used as input.
N = 10000, W = 128

could be tied to the 'groupDistanceCalc' function, which is designed to expedite the search process by pruning subtrees. In a larger tree, this function may inadvertently prune larger subtrees that could contain the correct answer, thereby compromising accuracy. Essentially, the pruning process is a trade-off between speed and precision, and as the tree expands, the likelihood of excluding the optimal subtree becomes higher, leading to a potential drop in the system's ability to correctly identify the most similar time series.

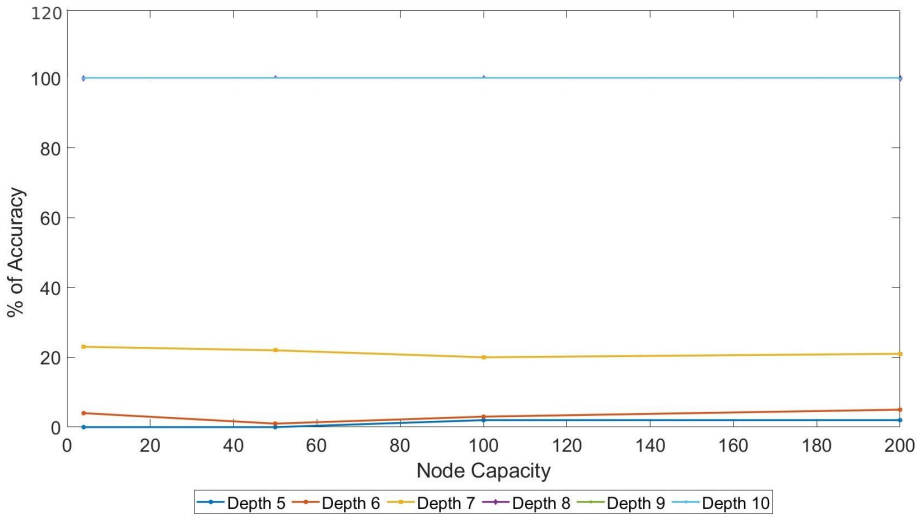
The outcomes from the second dataset demonstrate a notable improvement in accuracy compared to those from the custom dataset. This aligns with observations made with the HyperLogLog (HLL) based method. Such an increase in precision could be attributed to the inherent statistical characteristics of the time series data in the second dataset. Since the DSTree method relies heavily on the statistical attributes of the time series and their various segments and subsegments, the second dataset's properties may be more conducive to the DSTree algorithm, resulting in more accurate similarity estimations. This suggests that the second dataset's structure potentially aligns better with the method's operational dynamics, hence enhancing its effectiveness. Another explanation is be that, the similarity of time series within the custom dataset created for this study negatively influenced the accuracy of all three tested methods. This is evident as the data points were too akin to each other, complicating the task of accurately identifying similarities. On the contrary, the dataset provided by Amsaleg et al. contained time series with sufficient variance, which allowed the methods to more effectively distinguish and identify the correct similarities. The more distinct the time series, the easier it is for methods like HLL, Theta Sketches, and DSTree to perform accurate similarity estimations, highlighting the importance of dataset diversity in evaluating such algorithms.



(A) N = 10000, W = 128

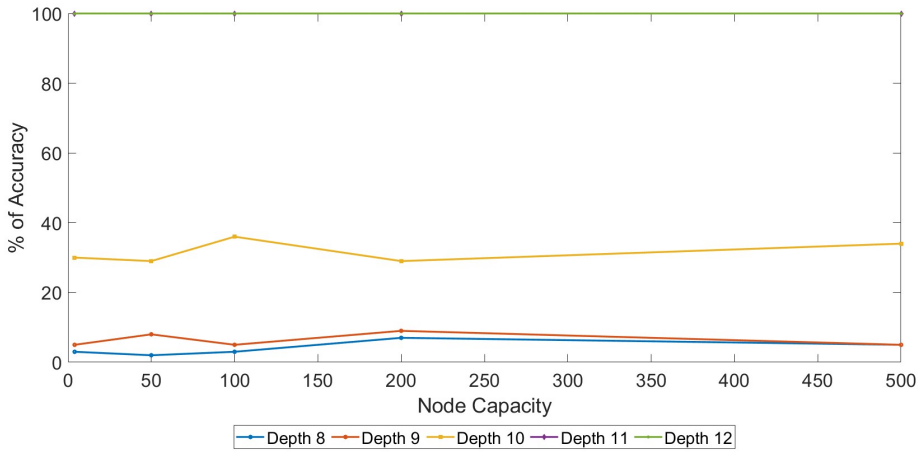


(B) N = 30000, W = 128

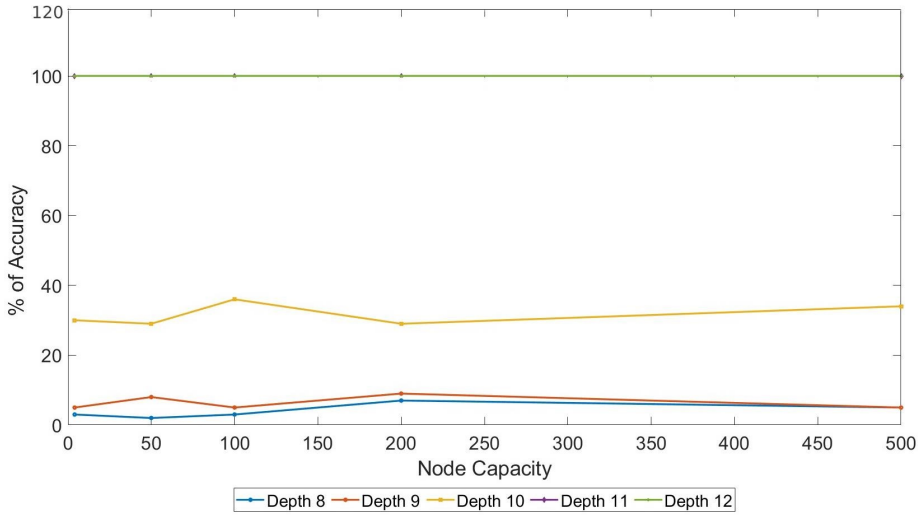


(C) N = 50000, W = 128

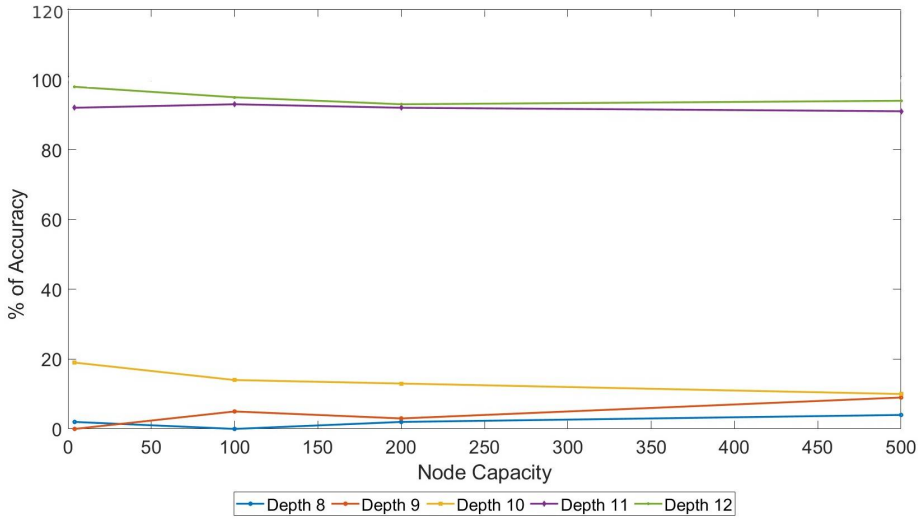
FIGURE 4.21: Accuracy of the DSTree based method, for window size



(A) N = 10000, W = 1024



(B) N = 30000, W = 1024



(C) N = 50000, W = 1024

FIGURE 4.22: Accuracy of the HLL based method, for window size equal to 1028 and N number of synopses.

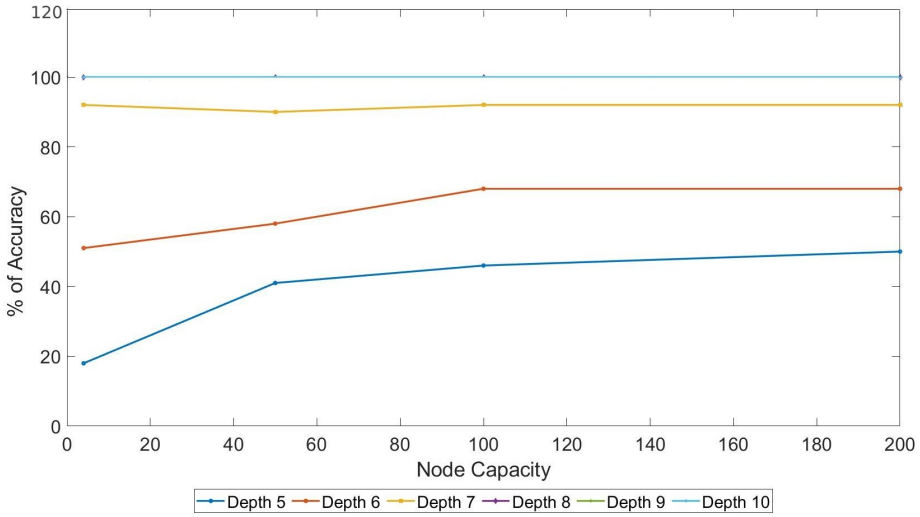


FIGURE 4.23: Accuracy results for the DSTree based method and where the dataset created by Amsaleg and Jégou was used as input. N = 10000, W = 128

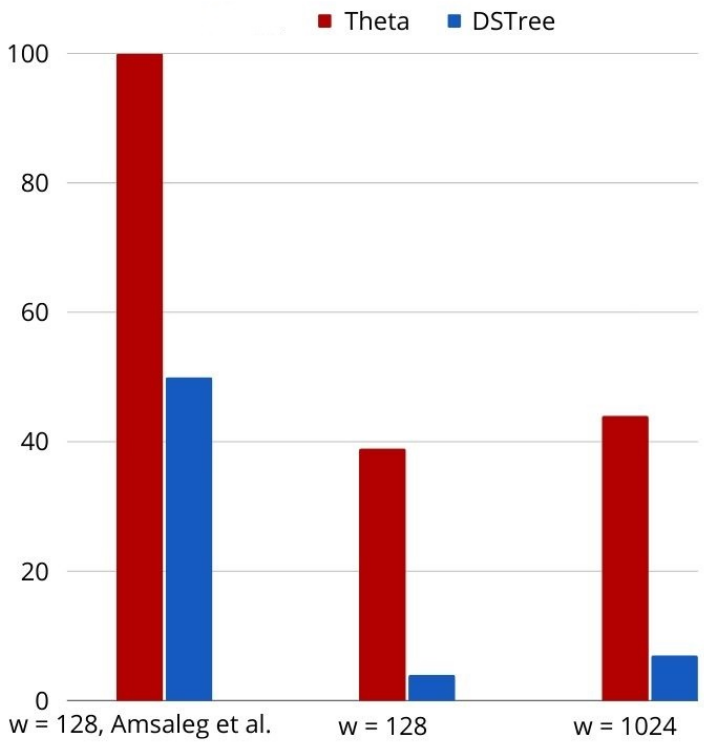


FIGURE 4.24: Comparative analysis of accuracy.

4.5 Comparing the Methods

Comparing the three methods — HyperLogLog (HLL), Theta Sketches, and DSTree — in terms of memory utilization and performance is a practical approach to evaluate their efficiency and accuracy. Given that each algorithm operates on distinct principles and offers different levers for accuracy and efficiency, standardizing the comparison based on memory usage is a reasonable strategy.

By configuring each method to generate synopses of equivalent size, we can control for memory use and focus on how each method utilizes this resource to maximize efficiency (throughput) and accuracy (correct similarity detection). For instance, HLL uses registers and hash functions, Theta Sketches adjust through the parameter k, and DSTree through tree depth and node capacity. Then the chosen configurations will be subjected to both of the available datasets, when the incoming windows are of size 128 and only in the custom dataset for windows of size 1024. Another important feature is for the synopsis to be of less size than the window, since otherwise the concept of synopsis becomes irrelevant.

In essence, this comparative analysis will highlight which method delivers the best trade-off between memory usage and the twin objectives of accuracy and efficiency. The findings will be particularly relevant for applications where memory is a constraint but performance cannot be compromised, such as processing streams of data in real-time.

The parameters of the HLL-based implementation significantly affect the memory usage and computational complexity of the algorithm. The '*number of registers*' parameter dictates the space complexity as it determines the count of registers utilized, which is $2^{(\text{number of registers})}$. Meanwhile, the '*number of hash functions*' parameter influences both space and time complexity, as it indicates how many sets of these registers are employed during the hashing process. For example, with parameters set to 5 for the number of registers and 2 for the number of hash functions, the algorithm would use $2^5 \times 2 = 64$ integers for its computation. The same memory usage can also be achieved with parameters set to 4 for the number of registers and 4 for the number of hash functions.

Calculations for memory allocation in Theta Sketch are more straightforward. The value of k directly corresponds to the number of registers used. To match the memory usage of other methods, k is set to 63. Along with an additional integer to store the theta value, the total memory requirement amounts to 64 integers. This simplicity in memory calculation allows for easier configuration of Theta Sketches when memory usage is a critical constraint.

For the DSTree, the memory requirement is determined by the depth of the tree. With a tree depth set to 5, the lowest level of the synopsis array has a size of 2^4, while the sizes for the higher levels decrease exponentially ($2^3, 2^2, \ldots$). Consequently, one array level requires 2^5 integers. Since DSTree utilizes two arrays — one for storing mean values of the segments and another for their standard deviation — the combined memory requirement totals $2 \times 2^5 = 2^6$ integers. This calculation ensures that the DSTree has sufficient memory allocation to store its structural components effectively. The node capacity will be set to 100 since from the earlier experiments we saw that this value maximizes the throughput, while at the same time is irrelevant to the accuracy of the method.

Given the outcomes from prior tests, where the HyperLogLog method's accuracy dropped to zero even when applying 256 registers for 128-sized windows, it's evident that reducing to a mere 32 registers would not yield viable results. As such,

the HyperLogLog method will be excluded from forthcoming comparative analyses, leaving just the Theta Sketches and DSTree methods for evaluation.

As illustrated in Figure 4.27, the comparative results of the three methods are presented in a unified plot for ease of analysis. Each method's performance is plotted to facilitate a direct comparison. This consolidated visualization aids in discerning the efficiency and efficacy of each method under the same memory constraints.

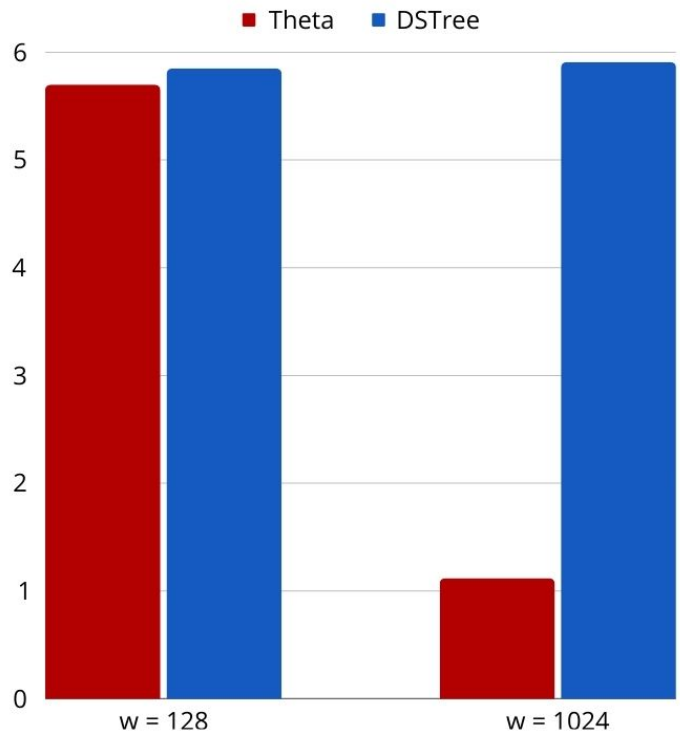


FIGURE 4.25: Comparative analysis of the synopsis creation throughput for the studied methods.

When examining smaller window sizes, it is evident that both methods are capable of meeting the output rate established by the Kafka pipeline. This suggests that they can generate synopses without causing a bottleneck, as their rate of creation is the same. However, the advantage of the DSTree method becomes apparent with larger window sizes. In such cases, DSTree maintains its level of performance, continuing to generate new synopses at the rate at which they arrive. In contrast, the throughput of Theta Sketches' synopsis creation falls significantly, managing to produce only a little more than one synopsis per second on average.

In assessing the query response speed of the two methods, DSTree demonstrates a clear superiority, processing queries nearly 50 times quicker than the Theta Sketches-based method. The disparity in performance becomes even more pronounced with larger window sizes, with DSTree delivering answers 167 times faster than Theta Sketches. This efficiency is due to the faster traversal possible in a tree-like structure to locate the closest match, as opposed to comparing the query against every synopsis. Additionally, the pruning capabilities of DSTree significantly enhance its speed, explaining the substantial gap in the response throughput between these methods.

Contrary to the previous metrics, DSTree falls significantly short in accuracy when compared to the Theta Sketch algorithm. As illustrated in Figure 4.27, even

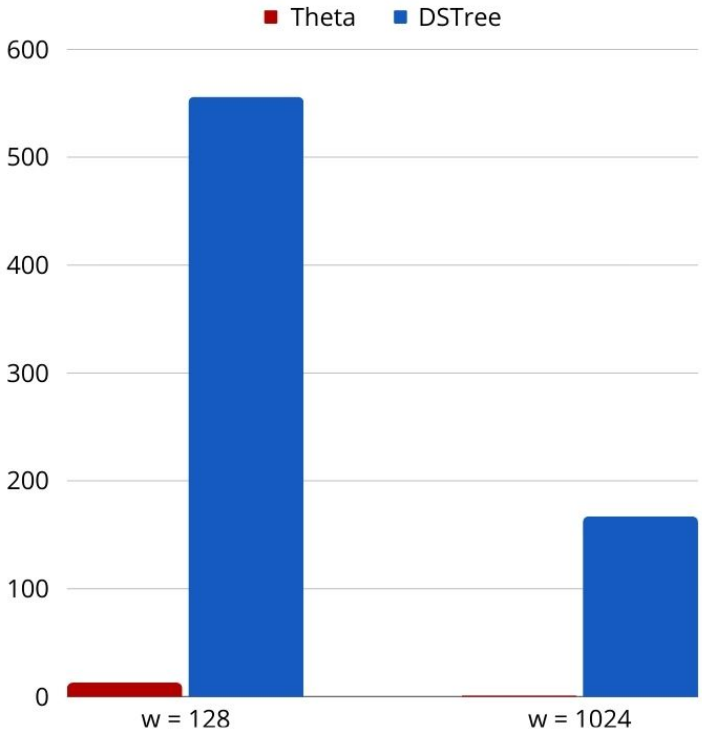


FIGURE 4.26: Comparative analysis of the query answering throughput for the studied methods.

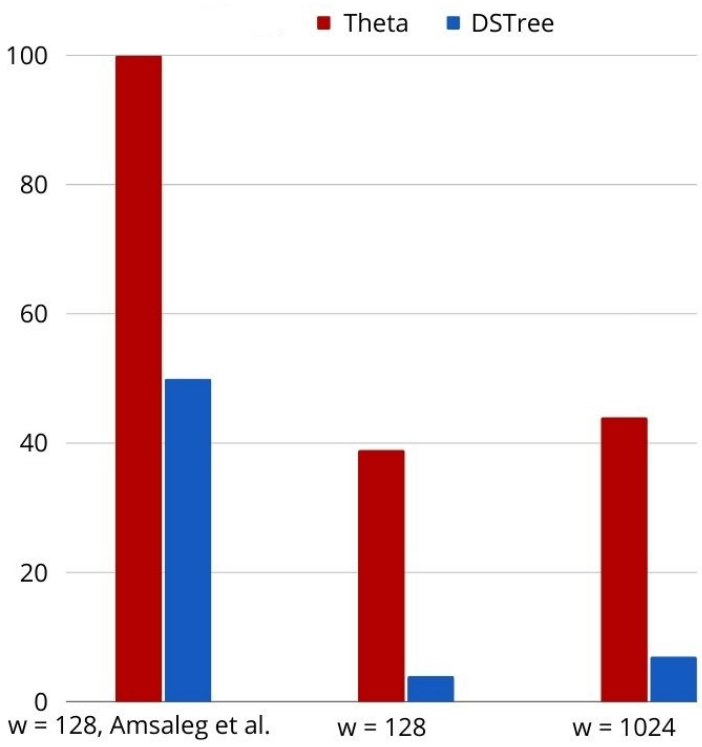


FIGURE 4.27: Comparative analysis of the accuracy for the studied methods when it comes to answering approximate similarity queries.

when leveraging the dataset provided by Amsaleg et al.—where DSTree performs at its best—it still achieves only half the accuracy of the Theta Sketch method. The disparity is even more pronounced with the custom dataset, particularly for smaller window sizes. Although there is a slight increase in DSTree's accuracy with larger windows, this improvement is marginal and overshadowed by the concurrent enhancement of Theta Sketch's results.

Chapter 5

Conclusions

The analysis of the three distinct methods—HyperLogLog (HLL), Theta Sketches, and DSTree—reveals clear trade-offs between memory usage, throughput, and accuracy. HLL, while advantageous in scenarios requiring less memory, does not perform well when restricted to a lower number of registers, as accuracy drops to zero. On the other hand, DSTree offers impressive throughput in both synopsis creation and query response, vastly outperforming Theta Sketches, especially as window sizes increase. However, this speed comes at a cost to accuracy, where Theta Sketches consistently outperforms DSTree, maintaining higher accuracy across varying datasets and window sizes.

These results suggest that the choice of method depends heavily on the specific requirements of the application. For tasks where speed is critical and approximate answers are acceptable, DSTree may be the preferred choice. Conversely, for applications where precision is paramount, Theta Sketches would be more appropriate, despite its slower performance.

Ultimately, this comparison underscores the importance of understanding the characteristics of the data and the priorities of the application. No one method emerges as superior in all aspects; hence, the decision must be based on a balanced consideration of the trade-offs involved.

5.1 Future Work

In contemplating the future trajectory of streaming data analysis, particularly in the context of methods like DSTree, HyperLogLog (HLL), and Theta Sketches, there emerges a landscape ripe with opportunities for enhancement and innovation.

One pivotal avenue is the scalability and handling of larger datasets. As the volume of data continues to expand, the need to efficiently process and analyze these vast datasets becomes increasingly critical. This necessitates not only improvements in existing methods but also the exploration of new algorithms that can scale effectively while maintaining, or even improving, accuracy and efficiency.

Another technical aspect that shows promise for future development is the optimization of these methods for varied datasets. The performance variability observed across different datasets suggests an opportunity for algorithms that can dynamically adapt based on the statistical properties of the data. This adaptability could significantly enhance the precision and applicability of these methods across diverse data streams.

Further, comprehensive comparative studies stand as a vital component of future work. There's a clear need for more in-depth and exhaustive comparisons between methods like HLL, Theta Sketches, and DSTree under various conditions. Such studies would uncover nuanced insights into each method's strengths, weaknesses, and optimal use cases, guiding practitioners in making informed choices.

Another technical frontier involves the development of automated configuration selection tools. These would employ algorithms or AI to recommend the most effective configurations for specific datasets and objectives, thereby streamlining the setup process and optimizing performance.

Continuing the technical exploration, expanding the range of datasets used in testing, particularly with real-world time series data from a variety of domains, is essential. This will not only validate the existing methods but also help in fine-tuning them to handle the complexities and peculiarities of different types of data.

In addition, the development of comprehensive, user-friendly tools and platforms that encapsulate these methods could significantly democratize access to advanced data analysis techniques. Making these methods accessible and easy to use for a broader audience can catalyze their adoption in diverse fields.

Lastly, comparing DSTree and similar methods with other prevalent techniques for estimating similarity in data streams, like MinHash, is crucial. This comparison would not only highlight the relative strengths and shortcomings of these methods but also pave the way for hybrid approaches that combine the best elements of different methods to achieve superior performance.

By focusing on these technical aspects, future research and development in the field of streaming data analysis can lead to significant advancements, making these methods more robust, versatile, and applicable across a wide array of real-world scenarios. This progress will be instrumental in harnessing the full potential of streaming data, driving innovation and insights in numerous domains.

Bibliography

[1] Pankaj Agarwal et al. "Mergeable Summaries". In: *ACM Transactions on Database Systems (TODS)* 38 (May 2012). DOI: 10.1145/2213556.2213562.

[2] *Apache Datasketches: The Theta Sketch Framework*. URL: https://datasketches.apache.org/docs/Theta.

[3] Daniel Baker and Ben Langmead. "Dashing: fast and accurate genomic distances with HyperLogLog". In: *Genome Biology* 20 (Dec. 2019). DOI: 10.1186/s13059-019-1875-0.

[4] Bar-Yossef et al. "Counting Distinct Elements in a Data Stream". In: Sept. 2002. ISBN: 978-3-540-44147-2. DOI: 10.1007/3-540-45726-7_1.

[5] Marc Bury and Chris Schwiegelshohn. "Efficient Similarity Search in Dynamic Data Streams". In: *IEEE Transactions on Knowledge and Data Engineering* PP (May 2016). DOI: 10.1109/TKDE.2019.2916858.

[6] Alessandro Camerra et al. "Beyond one billion time series: Indexing and mining very large time series collections with iSAX2+". In: *Knowledge and Information Systems* 39 (Apr. 2014). DOI: 10.1007/s10115-012-0606-6.

[7] Graham Cormode and Senthilmurugan Muthukrishnan. "An improved data stream summary: The Count-Min Sketch and its applications". In: *Journal of Algorithms* 55 (Apr. 2005), pp. 58–75. DOI: 10.1016/j.jalgor.2003.12.001.

[8] J. Gantz D. Reinsel and J. Rydning. "Data age 2025: The evolution of data to life-critical. Don't Focus on Big Data". In: (2017).

[9] Anirban Dasgupta et al. "A Framework for Estimating Stream Expression Cardinalities". In: (Oct. 2015).

[10] Fan Deng and Davood Rafiei. "Approximately detecting duplicates for streaming data using stable bloom filters". In: June 2006, pp. 25–36. DOI: 10.1145/1142473.1142477.

[11] Karima Echihabi et al. "Return of the Lernaean Hydra: experimental evaluation of data series approximate similarity search". In: *Proceedings of the VLDB Endowment* 13 (Nov. 2019), pp. 403–420. DOI: 10.14778/3368289.3368303.

[12] Otmar Ertl. "New cardinality estimation algorithms for HyperLogLog sketches". In: (Feb. 2017).

[13] Hakan Ferhatosmanoglu et al. "Vector Approximation based Indexing for Non-uniform High Dimensional Data Sets". In: *Proceedings of the 9th ACM International Conference on Information and Knowledge Management (CIKM), Washington, DC, USA* (Nov. 2000). DOI: 10.1145/354756.354820.

[14] Philippe Flajolet et al. "HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm". In: *Discrete Mathematics Theoretical Computer Science* DMTCS Proceedings vol. AH,... (Mar. 2012). DOI: 10.46298/dmtcs.3545.

[15] Tian Guo, Saket Sathe, and Karl Aberer. "Fast Distributed Correlation Discovery Over Streaming Time-Series Data". In: Oct. 2015, pp. 1161–1170. DOI: 10.1145/2806416.2806440.

[16] Stefan Heule, Marc Nunkesser, and Alexander Hall. "HyperLogLog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm". In: *ACM International Conference Proceeding Series* (Mar. 2013). DOI: 10.1145/2452376.2452456.

[17] *HuperLogLog: A probabilistic Data Structure*. URL: https://www.https://pangaj.github.io/HyperLogLog.

[18] T.s Jayram and D. Sivakumar. "Counting Distinct Elements in a Data Stream". In: (May 2003).

[19] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. "Product Quantization for Nearest Neighbor Search". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.1 (Jan. 2011), pp. 117–128. DOI: 10.1109/TPAMI.2010.57. URL: https://inria.hal.science/inria-00514462.

[20] Paulo Jesus, Carlos Baquero, and Paulo Almeida. "A Survey of Distributed Data Aggregation Algorithms". In: *Communications Surveys Tutorials, IEEE* 17 (Oct. 2011). DOI: 10.1109/COMST.2014.2354398.

[21] Kunio Kashino, Gavin Smith, and H. Murase. "Time-series active search for quick retrieval of audio and video". In: vol. 6. Apr. 1999, 2993 –2996 vol.6. ISBN: 0-7803-5041-3. DOI: 10.1109/ICASSP.1999.757470.

[22] D.E. Knuth. *The Art of Computer Programming: Fundamental algorithms*. Addison-Wesley series in computer science and information processing. Addison-Wesley Publishing Company, 1969. ISBN: 9780201038019. URL: https://books.google.gr/books?id=ZIlQAAAAMAAJ.

[23] Antonios Kontaxakis et al. "And synopses for all: A synopses data engine for extreme scale analytics-as-a-service". In: *Information Systems* 116 (May 2023), p. 102221. DOI: 10.1016/j.is.2023.102221.

[24] Flip Korn, Senthilmurugan Muthukrishnan, and Yihua Wu. "Modeling skew in data streams". In: June 2006, pp. 181–192. DOI: 10.1145/1142473.1142495.

[25] Rafał Kozik, Marek Pawlicki, and Michał Choraś. "A new method of hybrid time window embedding with transformer-based traffic data classification in IoT-networked environment". In: *Pattern Analysis and Applications* 24 (Nov. 2021). DOI: 10.1007/s10044-021-00980-2.

[26] Naama Kraus, David Carmel, and Idit Keidar. "Fishing in the Stream: Similarity Search over Endless Data". In: (Aug. 2017).

[27] Morris and Robert. "Counting large numbers of events in small registers". In: *Commun. ACM* 21 (Oct. 1978), pp. 840–. DOI: 10.1145/359619.359627.

[28] Rudi Poepsel Lemaitre et al. "In the Land of Data Streams where Synopses are Missing, the One Framework to Bring Them All". In: vol. 14. June 2021. DOI: 10.14778/3467861.3467871.

[29] Dennis Shasha. "Tuning Time Series Queries in Finance: Case Studies and Recommendations." In: *IEEE Data Eng. Bull.* 22 (Jan. 1999), pp. 40–46.

[30] *Theta Sketch Framework*. URL: \url{https://datasketches.apache.org/docs/Theta/ThetaSketchFramework.html}.

[31] Ashish Thusoo et al. "Hive - A Warehousing Solution Over a Map-Reduce Framework." In: *PVLDB* 2 (Aug. 2009), pp. 1626–1629. DOI: 10.14778/1687553.1687609.

[32] Ankit Toshniwal et al. "Storm@twitter". In: (June 2014). DOI: 10.1145/2588555.2595641.

[33] Machiko Toyoda, Yasushi Sakurai, and Toshikazu Ichikawa. "Identifying Similar Subsequences in Data Streams". In: Sept. 2008, pp. 210–224. ISBN: 978-3-540-85653-5. DOI: 10.1007/978-3-540-85654-2_23.

[34] Yang Wang et al. "A data-adaptive and dynamic segmentation index for whole matching on time series". In: *Proceedings of the VLDB Endowment* 6 (Aug. 2013), pp. 793–804. DOI: 10.14778/2536206.2536208.

[35] B. P. Welford. "Note on a Method for Calculating Corrected Sums of Squares and Products". In: *Technometrics* 4.3 (1962), pp. 419–420. DOI: 10.1080/00401706.1962.10490022. eprint: https://www.tandfonline.com/doi/pdf/10.1080/00401706.1962.10490022. URL: https://www.tandfonline.com/doi/abs/10.1080/00401706.1962.10490022.

[36] Huanmei Wu, Betty Salzberg, and Donghui Zhang. "Online Event-driven Subsequence Matching over Financial Data Streams." In: June 2004, pp. 23–34. DOI: 10.1145/1007568.1007574.

[37] Yunyue Zhu and Dennis Shasha. "Efficient Elastic Burst Detection in Data Streams". In: (Apr. 2003). DOI: 10.1145/956750.956789.

[38] Ariane Ziehn et al. "Time Series Similarity Search for Streaming Data in Distributed Systems". In: Mar. 2019.