TECHNICAL UNIVERSITY OF CRETE

Design and Implementation of a Distributed Synopsis Data Engine on Apache Flink

by

Antonis Kontaxakis

This thesis is submitted in partial fulfilment for M.Sc. degree in Computer Science

at the

School of Electrical and Computer Engineering

May 2020

ABSTRACT

This work, it details the design and structure of a Synopses Data Engine (SDE) which combines the virtues of parallel processing and stream summarization towards delivering interactive analytics at extreme scale. The SDE is built on top of Apache Flink and implements a synopsis-as-a-service paradigm. In that it achieves (a) concurrently maintaining thousands of synopses of various types for thousands of streams on demand, (b) reusing maintained synopses among various concurrent workflows, (c) providing data summarization facilities even for cross-(Big Data) platform workflows, (d) pluggability of new synopses on-the-fly, (e) increased potential for workflow execution optimization. The proposed SDE is useful for interactive analytics at extreme scales because it enables (i) enhanced horizontal scalability, i.e., not only scaling out the computation to a number of processing units available in a computer cluster, but also harnessing the processing load assigned to each by operating on carefully-crafted data summaries, (ii) vertical scalability, i.e., scaling the computation to very high numbers of processed streams and (iii) federated scalability i.e., scaling the computation beyond single clusters and clouds by controlling the communication required to answer global queries posed over a number of potentially geo-dispersed clusters.

ACKNOWLEDGEMENTS

First of all, I would like to thank Antonis Deligiannakis for his immediate and substantial assistance and his guidance during my thesis. Also, I would like to express my gratitude toward Nikos Giatrakos. The completion of this thesis would not be possible without his help.

Moreover, a big “thank you” to the other members of the examination committee, Mr. Vassilios Samoladas and Mr. Minos Garofalakis for their suggestions.

Finally a warn “thank you” to the fellow lab members, Arapi Xenia, the cluster administrator, my fellow colleagues Stamatakis Giorgos, Bitsakis Theodoris, Gkoutziouli Dimitra, Anesti Effrosyni, for their support, spotting mistakes and suggestions and my friends and family, who never stop caring.

This work has been partially funded from the European Union Horizon 2020 research and innovation Programme under Grant Agreement No 825070 (INFORE-PROJECT, http://www.infore-project.eu/).

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

Interactive extreme-scale analytics over massive, high speed data streams become of the essence in a wide variety of modern application scenarios. In the financial domain, NYSE alone generates several terra bytes of data a day, including trades of thousands of stocks [6]. Stakeholders such as authorities and investors need to analyse these data in an interactive, online fashion for timely market surveillance or investment risk/opportunity. In the life sciences domain, studying the effect of applying combinations of drugs on simulated tumours of realistic sizes can generate cell state data of 100 GB/min [25], which need to be analysed online to interactively determine successive drug combinations. In maritime surveillance applications, one needs to fuse high-velocity position data streams of hundreds of thousands of vessels across the globe and satellite, aerial images [31] of various resolutions. In all these scenarios, data volumes and rates are only expected to rise in the near future. In the financial domain, data from emerging markets, such as crypto currencies, are increasingly added to existing data sources. In life sciences, simulations are becoming progressively more complex, involving billions of interacting cells, while in the maritime domain autonomous vehicles are added as on-site sensing information sources. To enable interactive analytics at extreme-scale, stream processing platforms and systems need to provide three types of scalability:

• Horizontal scalability, i.e., the ability to scale the computation with extreme data volumes and data arrival rates as analysed in the aforementioned scenarios. This requires scaling out the computation to several machines and respective processing units available at a corporate data centre (cluster) or cloud. Horizontal scalability is achieved by parallelizing the processing and adaptively assigning computing resources to running analytics queries.

• Vertical scalability, i.e., the ability to scale the computation with the number of processed streams. For instance, to detect systemic risks in the financial scenario, i.e., stock level events that could trigger instability or collapse of an entire industry or economy, requires discovering and interactively digging into correlations among tens of thousands of stock streams. The problem involves identifying the highly correlated pairs of stock data streams under various statistical measures, such as Pearson's correlation over N distinct, high speed data streams, where N is a very large number. To track the full Θ(N^2) correlation matrix results in a quadratic explosion in space and computational complexity which is simply infeasible for very large N. The problem is further exacerbated when considering higher-order statistics (e.g., conditional dependencies/correlations). The same issue arises in the maritime surveillance scenario for trajectory similarity scores over hundreds of thousands of vessels. Clearly, techniques that can provide vertical scaling are sorely needed for such scenarios.

• Federated scalability, i.e., the ability to scale the computation in settings where data arrive at multiple, potentially geographically dispersed sites. On the one hand, a number of benchmarks [[29], [33]] conclude that, in such settings, even if horizontal scalability is ensured within each cluster, the maximum achieved throughput (number of streaming tuples that are processed per time unit) is network bound. On the other hand, consider again the systemic risk detection scenario from the financial domain where stock trade data arrive at geo-dispersed Data Centres around the globe. Moving entire data streams around the sites in order to extract pairwise correlation scores depletes the available bandwidth, introducing network latencies that prevent the interactivity of the desired analytics.

Big Data platforms, including Apache Flink [[2]], Spark [[4]], Storm [[5]] among others, have been developed that support or are especially dedicated to stream processing. Such platforms focus on horizontal scalability, but they are not sufficient by themselves to allow for the required vertical and federated scalability. On the other hand, there is a wide consensus in stream processing [[17], [19], [23], [30], [34]] that approximate but rapid answers to analytics tasks, more often than not, suffice. For instance, knowing in real-time that a group of approximately 50 stocks, extracted out of thousands or millions of stock combinations, is highly (e.g., > 0.9 score) correlated is more than enough to detect systemic risks. Therefore, such an approximate result is preferable compared to an exact but late answer which says that the actual group is composed of 55 stocks with correlation scores accurate to the last decimal. Data summarization techniques such as samples, sketches or histograms [17] build carefully crafted synopses of Big streaming Data which preserve data properties important for providing approximate answers, with tuneable accuracy guarantees, to a wide range of analytic queries. Such queries include, but are not limited to, cardinality, frequency moment, correlation, set membership or quantile estimation [17]. Data synopses enhance the horizontal scalability provided by Big Data platforms. This is because parallel versions of data summarization techniques, besides scaling out the computation to a number of processing units, reduce the volume of processed high-speed data streams. Hence, the complexity of the problem at hand is harnessed and execution demanding tasks are severely sped up. For instance, sketch summaries [18] can aid in tracking the pairwise correlation of streams in space/time that is sublinear in the size of the original streams. Additionally, data synopses enable vertical scalability in ways that are not possible otherwise. Indicatively, the coefficients of Discrete Fourier Transform (DFT)-based synopses [34] or the number of set bits (a.k.a. Hamming Weight) in Locality Sensitive Hashing (LSH)-based bitmaps [26] have been used for correlation aware hashing of streams to respective processing units. Based on the synopses, using DFT coefficients or Hamming Weights as the hash key respectively, highly uncorrelated streams are assigned to be processed for pairwise comparisons at different processing units. Thus, such comparisons are pruned for streams that do not end up together. Finally, federated scalability is ensured both by the fact that communication is reduced since compact data stream summaries are exchanged among the available sites and by exploiting the mergeability property [11] of many synopses' techniques. As an example, answering cardinality estimation queries over several sites, each maintaining its own FM sketch [17] is as simple as communicating only small bitmaps (typically 64-128 bits) to the query source and performing a bitwise OR operation. In this work, we detail the design and structure of a Synopses Data Engine (SDE) built on top of Apache Flink ingesting streams via Apache Kafka [3]. Our SDE combines the virtues of parallel processing and stream summarization towards delivering interactive analytics at extreme scale by enabling enhanced horizontal, vertical and federated scalability as described above. However, the proposed SDE goes beyond that. Our design implements a Synopsis as-a-Service (termed SDEaaS) paradigm where the SDE can serve multiple, concurrent application workflows in which each maintained synopsis can be used as an operator. That is, our SDE operates as a single, constantly running Flink job which achieves:

- Concurrently maintaining thousands of synopses for thousands of streams on demand
- Reusing-maintained synopses among multiple application workflows (submitted jobs) instead of redefining and duplicating streams for each distinct workflow separately
- Pluggability of new synopses' definitions on-the-fly
- Providing data summarization facilities even for cross-(Big Data) platform workflows outside of Flink
- Optimization of workflows execution by enabling clever data partitioning
- Advanced optimization capabilities to minimize workflow execution times by replacing exact operators (aggregations, joins etc) with approximate ones, given a query accuracy budget to be spent

1.1 Motivation & Thesis Contribution

Few prior efforts provide libraries for online synopses maintenance, but neglect parallelization aspects [8], [9], or lack a SDEaaS design [7] needing to run a separate job for each maintained synopsis. The latter compromises aspects in points A-F above and increases cluster scheduling complexity. Others [32] lack architectural provisions for federated scalability and are limited to serving simple aggregation operators being deprived of vertical scalability features as well. On the contrary, our proposed SDE not only includes provisions for federated scalability and provides a rich library of synopses to be loaded and maintained on the fly, but also allows to plug-in external, new synopsis definitions customizing the SDE to application field needs. More precisely, this thesis contribution is:

1. It presents the novel architecture of a Synopses Data Engine (SDE) capable of providing interactivity in extreme-scale analytics by enabling various types of scalability
2. The SDE is built using a SDE-as-a-Service (SDEaaS) paradigm, it can efficiently maintain thousands of synopses for thousands of streams to serve multiple, concurrent, even cross-(Big Data) platform, workflows.
3. It describes the structure and contents of our SDE Library, the implemented arsenal including data summarization techniques for the proposed SDE, which is easily extensible by exploiting inheritance and polymorphism.

4. It presents a detailed experimental analysis using real data from the financial domain to prove the ability of our approach to scale at extreme volumes, high number of streams and degrees of geo-distribution, compared to other candidate approaches.

1.2 Outline

Chapter 2 provides useful background related to concepts used throughout the thesis, while Chapters 3 describes the design and the functionality of the SDE, Chapter 4 present our implementation in detail, the architecture and gives insides of how each part fulfils its role. Chapter 5 analyses the performance of our SDE. Finally, Chapter 6 gives some closing remarks along with interesting directions for future.

2 BACKGROUND

This chapter includes all the necessary theoretical background knowledge that is essential to understand the thesis and provides a small briefing of the tools that was used to implement the SDE.

2.1 Data Stream Management

In the traditional data science world an analyst is provided with a given array/list/table or even a graph of finite number of elements/nodes/tuples, in other words a dataset, and he is assigned to solve a problem, providing value to his organization. This idea builds on the concept that the datasets are stored in a defined storage, they are available at all time and can be queried/updated many times in their lifetime. However, anyone can easily show that this idea cannot be applied to a variety of applications. When data become an infinite source of knowledge the need for algorithms that operate in an infinite continuous stream of data becomes a must. Every event-driven application, where data are treated as events that can trigger actions, for example, the temperature measurement passed a high enough threshold, a ship vessel has deviate from his normal route, can't be mapped to a traditional database application. These applications that monitor data, detect (anomalies), that run with timestamped data (events) should run in an online streaming fashion. This different idea may sound simple, but it needs a change of mindset. The basic principles of a traditional Database management system (DBMS) and those of a Data stream management system (DSMS) are different. Just to name a few in a DBMS you can assume unlimited storage cause, it can use hard discs, but on the other hand DSMS everything should run on a limited main memory, the state of the DSMS, so it can support the potentially extremely high update rate, when in DBMS the relative update rate is lower. On other difference is that DBMS provide mostly one-time queries that run thought the whole datasets and provide an exact answer but in DSMS most queries are static and continuous provides an answer in a data or time driven window. As shown in the Figure 1.

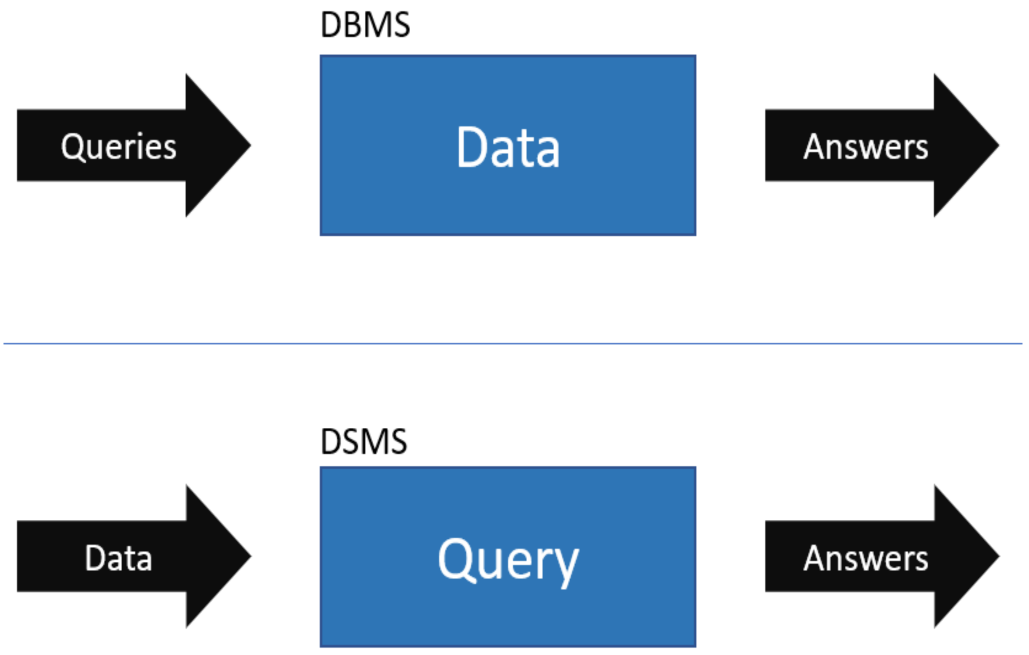


Figure 1: Difference Between DBMS and DSMS

But the DSMS and DBMS aren't limited to exact answer, be accepting a bounded error to the accuracy of the answer a data management system can provide the user, with a Synopsis that can be queried and support more than static, exact queries.

2.2 Data Summarization

In the world of big data there are often queries that cannot be scaled well, they require huge computational resources and time to provide an exact solution. Examples of those quires are count distinct items, most frequent items, exact joins, cardinality and many more. Of example to get the most frequent items to get the exact answer someone would need to keep track of each items and how many times it has seen it, that requires O(N) space complexity, which translates to a counter for each data point, billions of counters in a real big data scenario. That is where a specialized class of algorithms, which has many names, sketches, streaming algorithms, probabilistic data structures, or Synopses that can provide an approximated answer. By using the same example to get the most frequent items using a CountMinSketch with roughly one thousand (1024) counters the query can be answered with a relative error of approximately 1.5 percent with a probability of 99.6 percent.

This thesis would use the terminology Synopsis. Synopses are a class of data structures, that have the property to represent, summarize or even compress extremely large data sets in logarithmic or constant space complexity.

Given this definition any of the following can be consider a synopsis:

- Any Sampling method
- Histograms and quantiles
- Any Mean (AM, GM, HM)
- BloomFilter
- CountMinSketch

Even more complex transformations like DFT, wavelets or coresets can be viewed as synopsis. Synopses can provide answers orders-of magnitude faster and many of them with a mathematically proven error bound. Also, one property that is many times overlooked is the diversity that using a synopsis can provide. Synopses can produce answers to one or more queries, by keeping track of any sampling Synopsis, the system can provide estimation of all aggregation queries and many more and that’s one-way interactivity can be achieved. As stated by Beaudouin- Lafon “An interactive system is a computer application that takes into account, during its execution, information communicated by the user or users of the system, and which produces, during its execution, a perceptible representation of its internal state”. For creating an interactive DSMS there may not be other viable alternatives, and in the case of real-time analysis, synopses are the only known solution.

Below we outline the main and some highly desirable properties, so any data structure that apply to these properties can be characterized as a Synopsis:

Synopses Characteristics:

1. **One Pass**: the synopses are easily can be created, during an exactly one-pass over the streaming data in the (arbitrary) order of their arrival.
2. **Small Update Time**: the time it requires for the updates of the structure to take place slow be extremely low to be able to support the extreme rate of the input, desirable is O(1) and in worst case poly-logarithmic in N.
3. **Small Memory footprint**: the requirements for memory should be bounded and small O(K) where K<<N (especial for the a system like SDE that supports multiple synopses,

for multiple datasets, bounded memory usage is a must to guarantee that one synopsis don’t over utilized resources). Algorithms with poly-logarithmic in N space requirements can be used also but with memory concerns.

In addition, three highly desirable properties for stream synopses are:

1. **Delete-proof**: the data structure can handle both insertions for new data and deletions for expiring data
2. **Provide Error guarantees**: Providing a user with a fixed error bound so he can account for the loss in accuracy, which is crucial in many scenarios
3. **Composable**: (An important feature in this distributed system) the synopsis can be built on a distributed way on different instances of the system, where each instance gets a part of the stream. Then the answer to a query can be obtained by composing the answer of each synopsis instance and applying a reduce function or by merging the distributed synopsis instances in a simple (and, ideally, lossless) fashion to obtain a synopsis of the entire stream.

In the sections below we describe the synopses that are currently supported by the SDE, but as it is shown in later chapters, the SDE isn’t limited to these, can easily be configured to support any data structure that apply to the above properties.

2.2.1 Discrete Fourier Transform (DFT)

Our DFT-based correlation estimation implementation is mostly based on StatStream [34]. We note beforehand that there is a direct relation between Pearson's correlation coefficient among time series x, y and the Euclidean distance of their corresponding normalized version (we use primes to distinguish DFT coefficients of normalized time series from the ones of the unnormalized version). In particular, $Corr(x, y) = 1 - \frac{1}{2}d^2(X', Y')$, where *d(.)* is the Euclidean distance.

The Discrete Fourier Transform transforms a sequence of N complex numbers $x_0,,x_{N-1}$ into another sequence of complex numbers $X_{0},\ldots,X_{N-1}$, which is defined by the DFT Coefficients, calculated as:

$$X_F = \frac{1}{N}\sum_{k=1}^{N-1} x_k \, e^{\frac{i2\pi kF}{N}}, \text{ for F=0..., N-1 and } i = \sqrt{-1}$$

Compression is achieved by restricting F in the above formula to few coefficients. In our implementation we set *F=0,….,7*, i.e., we use up to 8 coefficients for comparing time series, after performing an exploratory analysis on the time series present in INFORE-related datasets and because of the fact that the majority of the Fourier signal energy is concentrated on the first few coefficients [35].

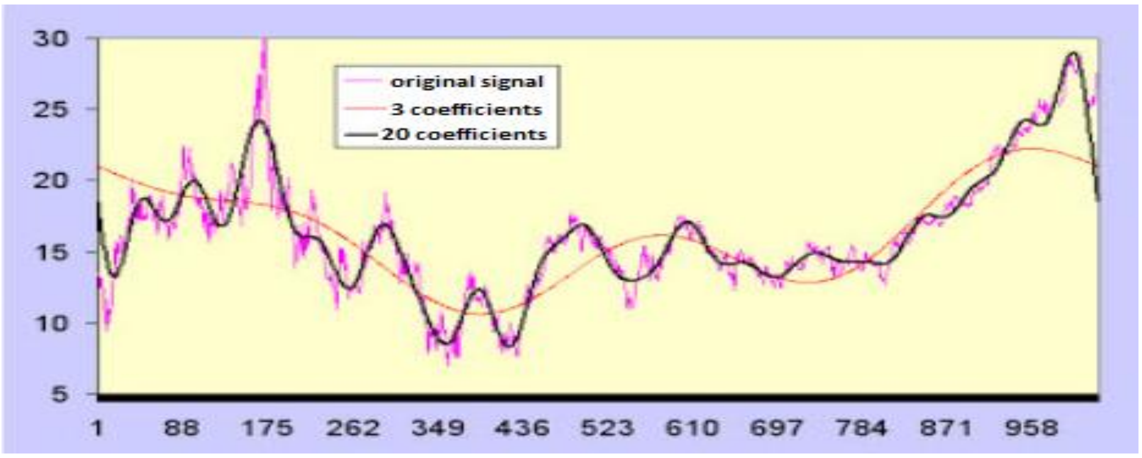


Figure 2: Approximating time series with DFT using 3 and 20 [36] Coefficients

There is a couple of additional properties of the DFT which we take into consideration for computing time series similarities using the DFT instead of the original time series and for parallelizing the processing load of pairwise comparisons among time series, respectively:

- The Euclidean distance of the original time series and their DFT is preserved. We use this property to estimate the Euclidean distance of the original time series using their DFTs.
- It holds that: $Corr(x, y) \geq 1 - \varepsilon^2 \Rightarrow d((X'), (Y')) \leq \varepsilon$. This says that it is meaningful to examine only pairs for time series for which $d((X'), (Y')) \leq \varepsilon$. We use this property to bucketize (hash) time series based on the values of their first coefficient(s) and then assign the load of pairwise comparisons within each bucket to processing units working in parallel.

The DFT Coefficients can be updated incrementally upon operating over sliding windows. Assuming a window of size *w*, with a slide size *b*, then for the *F*-th coefficient we have:

$$X_F^{new} = e^{\frac{i2\pi bF}{w}} X_F^{old} + \frac{1}{N}\left(\sum_{k=0}^{b-1} x_{w+k}\, e^{\frac{i2\pi F(b-i)}{w}} - \sum_{k=0}^{b-1} x_k\, e^{\frac{i2\pi F(b-i)}{w}}\right)$$

Thus, in order to update the coefficients based on the window, as new tuples arrive, we have to keep for each coefficient that we decide to include in our approximation, the quantities $\sum_{k=0}^{b-1} x_k e^{\frac{i2\pi F(b-i)}{w}}$ (for the F-th coefficient).

Let us now explain how the time series that are approximated by the DFT coefficients are bucketized so that possibly similar time series are hashed to the same or neighbouring buckets, while the rest are hashed to distant buckets and, therefore, they are never compared for similarity. Beforehand, we note that the idea is that, in our parallel setting, buckets are assigned to different processing units which undertake the load of pairwise time series comparisons. Time series that are hashed to more than one buckets are replicated an equal amount of times.

Now, assume a user-defined threshold $T \in [0, 1]$. According to our above discussion, in order for the correlation to be greater than *T*, then *d(X',Y')* needs to be lower than ε, with $T = 1 - \varepsilon^2$. By using the DFT on normalized sequences, the original sequences are also mapped into a bounded feature space. The norm (the size of the vector composed of the real and the imaginary part of the complex number) of each such coefficient is bounded by $\frac{\sqrt{2}}{2}$.

Based on the above observation, [34] claims that the range of each DFT coefficient is between $-\frac{\sqrt{2}}{2}$ and $\frac{\sqrt{2}}{2}$. Therefore, the DFT feature space is a cube of diameter $\sqrt{2}$. Based on this, we use a number of DFT coefficients to define a grid structured, composed of cells/buckets for hashing groups of time series to each of them. Each cell is the grid is of diameter ε and there are in total $2\left\lceil\frac{\sqrt{2}}{2\varepsilon}\right\rceil^{\#used\_coefficients}$ buckets.

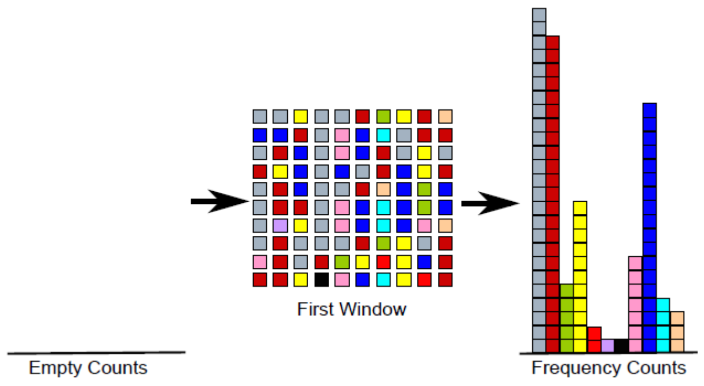
Each time series is hashed to a specific bucket/cell inside the grid. Suppose *X'* is hashed to bucket (c_1, c_2, ..., c_7). To detect the time series whose correlation with *X'* is above T, only time series hashed to adjacent cells are possible candidates. Those time series are a super-set of the true set of highly correlated ones. Since the cell diameter is ε and we use up to 7 coefficients for indexing (we can even keep up to 7, but use fewer for bucketizing the time series), time series mapped to non-adjacent cells possess a Euclidean distance greater than ε, hence, their respective correlation is guaranteed to be lower than *T*. Moreover, due to that property there will be no false negative comparisons. On the contrary there will be false positives which are eliminated by computing the pairwise distance among the DFTs of the hashed time series per bucket.

Note that the principal role of the SDE component is to produce the corresponding DFT coefficients and hash them to buckets. Therefore, the output of the corresponding synopsis in Figure 17 includes the resulted coefficients and the bucket identifier. The actual similarity tests (in each bucket) may be performed by some downstream operator in the designed workflow or the SDE itself as a special Reduce function 4.6.2.

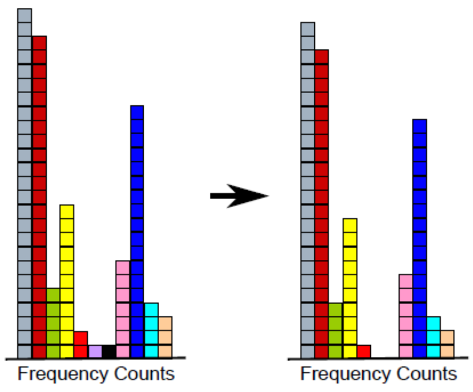
2.2.2 Lossy Counting

The Lossy Counting algorithm maintains a data structure, which is a set of entries of the form *<element, frequency, ε>*, where *element* is a data element, *frequency* is an integer representing the estimated frequency of the element and *ε* tunes the allowed maximum possible error in frequency estimation. According to the Lossy Counting rationale [38],

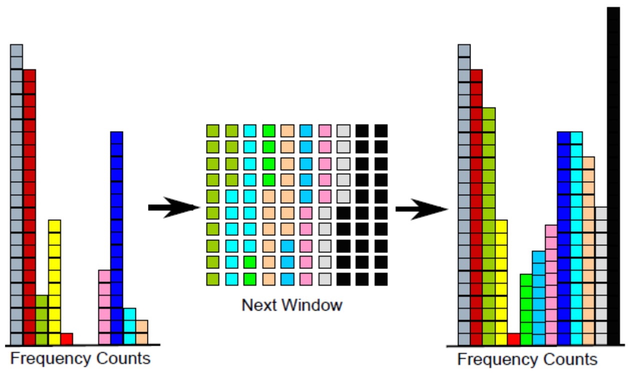
incoming data are conceptually divided into windows (often termed buckets) of $w = \left\lceil \frac{1}{\varepsilon} \right\rceil$ tuples each. At the beginning all maintained counters are empty. When an update *element* arrives for which a counter is already maintained, the corresponding *frequency* is increased by one. In case *element* is not monitored, a new counter entry is created. Then, at the end of the window all counters are reduced by one, while if a counter becomes zero for a specific *element*, it is dropped. All running counters can be maintained in a HashTable-like data structure. If *N* tuples have streamed in so far, given *ε* and a support parameter *s* such that ε=0.1s: (a) all items whose true frequency exceeds *sN* are output, (b) no elements whose true frequency is less than *(s-ε)N* are output, (c) the estimated frequencies are less than the true frequencies by at most *εN*. For instance, for *s = 10%, ε = 1%* and *N = 1000*, all elements exceeding a frequency of 100 will be included in the output stream, which will contain no elements with frequencies below 90. In this example, all estimated frequencies diverge from the true frequencies by at most 10. Finally, false positive frequent elements for frequencies between 90 and 100 might or might not be included in the output stream.



(a) First window of w size



(b) At the end of the window reduce all counters by 1

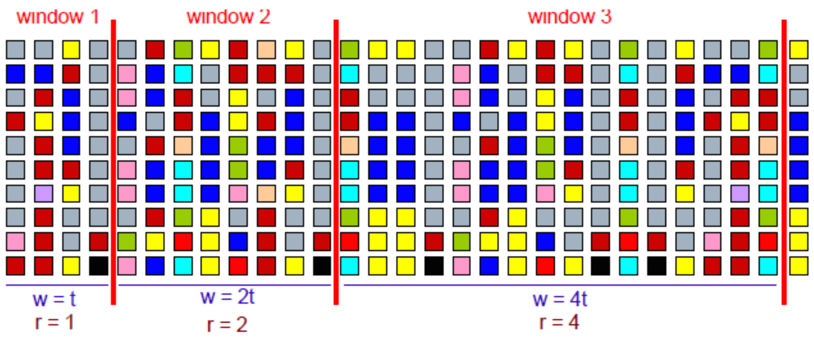


(c) Process next window of size w

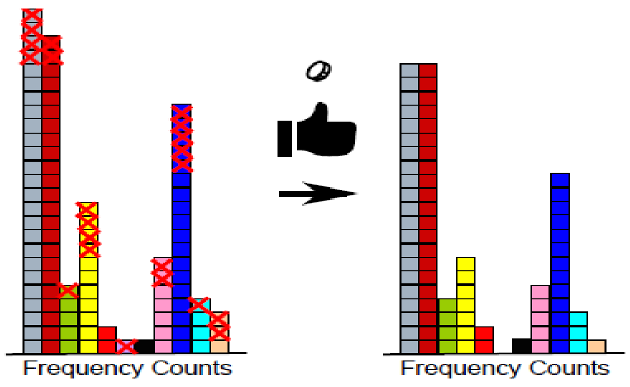
Figure 3: Lossy Counting over a window of colours

2.2.3 Sticky Sampling

Sticky Sampling [38] shares similarities with Lossy Counting, but differs in that (i) the size of buckets/windows is not steady and (ii) the count of an element is maintained with a certain sampling probability. This introduces a probability δ for the estimated frequencies to be less than the true frequencies by more than *εN*. The input of the algorithm is composed of a support parameter s and the desired *(ε,δ)* guarantees. The algorithm works as follows: the incoming stream is split into windows. The first window is of $w = t = \frac{1}{\varepsilon}\log\left(\frac{1}{s\delta}\right)$ size and this size is doubled in each subsequent one, i.e., $w = 2t$, $w = 4t$ and so on. For each window, Sticky Sampling goes through elements and if a counter for an element exists, it increases it by one each time the element is observed. If a counter for an element does not exist, one is created and initialized to 1 with probability $\frac{t}{w}$. At the end of each window, for each element's counter a coin is tossed. If the result of the coin flip is 0 the counter is reduced by one. Otherwise, the processed continuous by flipping a coin for the counter of the next element. If a counter of an element becomes zero, the element is dropped. The guarantees mentioned for Lossy Counting also hold for Sticky Sampling, except for a failure probability δ as described in point (ii) above.



(a)Dynamic window size. Sampling rate grows in proportion to window size



(b) At the end of the window, for each counter toss coin – adjust counts.

Figure 4: Sticky Sampling over a window of colors

2.2.4 Counion Sketch

A Count-Min Sketch [19] is a two-dimensional array of $w \times d$ dimensionality used to estimate frequencies of elements of a stream using limited amount of memory. For given accuracy ε and error probability δ, $w = \frac{e}{\varepsilon}$ and $d = log\frac{1}{\delta}$. d random, pairwise independent hash functions are chosen for hashing each element to a column in the sketch. When an element streams in, it goes through the d hash functions so that one counter in each row is incremented. The estimated frequency for any item is the minimum of the values of its associated counters. This provides an estimation within εN with probability at least 1-δ. For each new element, if its frequency is greater than a required threshold, it is added to a heap. At the end, all elements whose estimated count is still above the threshold are output. In the implemented version of CountMin sketch no such threshold is employed, thus every counter of a queried element is outputted.

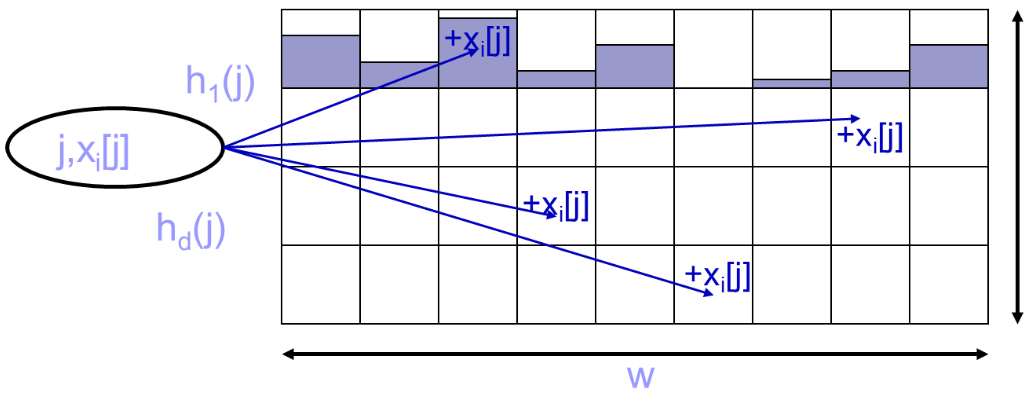


Figure 5: CM Sketch Structure. Each entry in vector x is mapped to one bucket per row. Two sketches can be merged via entry-wise summation. x_i[j] is estimated by taking $\min_k$sketch[k,h_k(j)][39].

2.2.5 Bloom Filters

A Bloom filter [14] is a space-efficient representation of a stream of *n* elements from a universe *U*, mainly used to deduce whether a certain element has been observed (set membership). A Bloom filter is a bitmap of *m* bits equipped with a family of *k* independent hash functions which hash stream elements to positions of the bitmap. The whole bitmap is initially unset. An element is inserted into the Bloom filter by setting all positions of the bitmap where the *k* hash functions point to, to 1. At query time, an element is assumed to be contained in the original set of observations if all hashed positions of the Bloom filter are equal to 1. If at least one of these positions is set to 0, then we conclude that the element is not present.

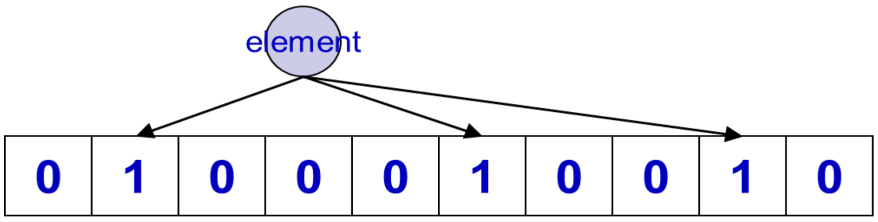


Figure 6: Bloom filter operation. k=3 hash functions map items to bit vector k times, set all k entries to 1 to indicate item is present [39].

Bloom filters exhibit a small probability of false positives; due to hash collisions, it is possible that all bits representing a certain element have been set to 1 by the insertion of other elements.

The probability for a false positive is ~ $(1 - e^{-kn/m})^k$. For given n and Bloom filter length, the false positive probability can be minimized by optimizing the ratio between true bits and Bloom filter length. We denote this ratio as Bloom filter density. The false positive probability is minimized when this density is 0.5. This is the case when the number of hash functions is set to $k \approx m/n\ ln(2)$. Equivalently, for given n, false positive probability and $k \approx m/n\ ln(2)$ we can compute the required size of the Bloom filter.

Set membership in case of union or intersection of separate sets of streaming elements can be assessed by performing bitwise OR and AND operations respectively, on the corresponding bitmaps.

2.2.6 HyperLogLog Sketch

The HyperLogLog algorithm constitutes the evolution of FM sketches [22] and the LogLog algorithm [38]. It is a simple, elegant algorithm that enables to extract distinct counts using limited memory and a simple error approximation formula. In the common implementation of HyperLogLog, each incoming element is hashed to a 64-bit bitmap. The hash function is designed so that the hashed values closely resemble a uniform model of randomness, i.e., bits of hashed values are assumed to be independent and to have ½ probability of occurring each.

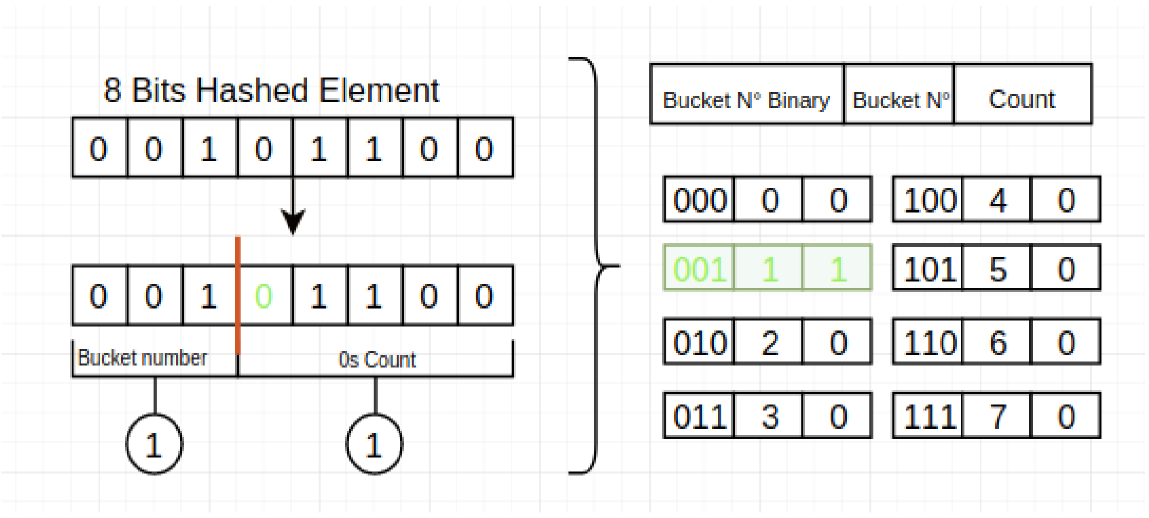


Figure 7: HyperLogLog maintenance example

The first m bits of the bitmap are used for bucketizing the element and we have an array M of 2^m buckets (also called registers). The rest $64\text{-}m$ bits are used so as to count the number of leading zeros and in each bucket, we store the maximum such number of leading zeros to that particular bucket. To extract a distinct count estimation, we average the values of the buckets. The relative error of HLL in the estimation of the distinct count is $1/\sqrt{2^\wedge m}$. HLL are trivial to merge based on the above procedure and equivalent number of buckets maintained independently.

2.2.7 AMS Sketch

The key idea in AMS sketch [12] is to represent a streaming (frequency) vector *v* using a much smaller sketch vector *sk(v)* that is updated with the streaming tuples and provide probabilistic guarantees for the quality of the data approximation. The AMS sketch defines the *i*-th sketch entry for the vector *v*, *sk(v)[i]* as the random variable $\sum_k v[k] \cdot \xi_i[k]$, where $\{\xi_i\}$ is a family of four-wise independent binary random variables uniformly distributed in *{−1, +1}* (with mutually-independent families across different entries of the sketch). Using appropriate pseudorandom hash functions, each such family can be efficiently constructed on-line in logarithmic space. Note that, by construction, each entry of *sk(v)* is essentially a randomized linear projection (i.e., an inner product) of the *v* vector (using the corresponding *ξ* family), that can be easily maintained (using a simple counter) over the input update stream. Every time a new stream element streams in, we just have to add $v[k] \cdot \xi_i[k]$ to the aforementioned sum and similarly for element deletion – expiration. Each sketch vector can be viewed as a two-dimensional $n \times m$ array (Buckets and Depth in), where $n = O(\frac{1}{\varepsilon^2})$, and $m = O(log\frac{1}{\delta})$, with ε, 1 − δ being the desired bounds on error and probabilistic confidence, correspondingly. The “inner product” in the sketch-vector space for both the join and self-join case (in which case we replace $sk(v_2)$ with $sk(v_1)$ in the formula below) is defined as:

$$sk(v_1) \cdot sk(v_2) = \underbrace{median}_{j=1,\dots,m} \left\{ \frac{1}{n} \sum_{i=1}^{n} sk(v_1)[i,j] \cdot sk(v_2)[i,j] \right\}$$

Furthermore, AMS sketches can capture several other interesting query classes, including range and quantile queries, heavy hitters, top-k queries, approximate histogram, and wavelet representations.

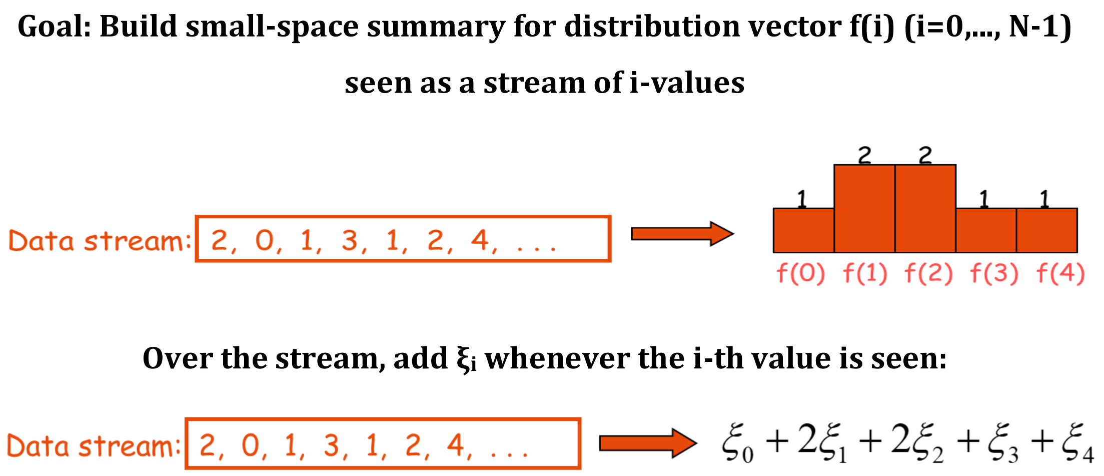

Goal: Build small-space summary for distribution vector f(i) (i=0,..., N-1) seen as a stream of i-values
Data stream: 2, 0, 1, 3, 1, 2, 4, . . .
1
2
2
1
1
f(0)
f(1)
f(2)
f(3)
f(4)
Over the stream, add ξi whenever the i-th value is seen:
Data stream: 2, 0, 1, 3, 1, 2, 4, . . .
$\xi_0 + 2\xi_1 + 2\xi_2 + \xi_3 + \xi_4$

Figure 8: AMS Sketch computation [40]

Another important property is the linearity of AMS sketches: Given two “parallel” sketches (built using the same ξ families) *sk(v₁)* and *sk(v₂)*, the sketch of the union of the two underlying streams (i.e., the streaming vector $v_1 + v_2$) is simply the component-wise sum of their sketches; that is, $sk(v_1 + v_2) = sk(v_1) + sk(v_2)$.

2.2.8 Chain Sampling

The chain sampling algorithm [13] provides a simple random sample without replacement of size *k* over a sliding window of *n* cardinality where *k* is expected to be much lower than *n*. We here describe the algorithm for sampling a single element from the sliding window. In order to obtain a sample of *k* size, this process needs to be repeated *k* times for each element.

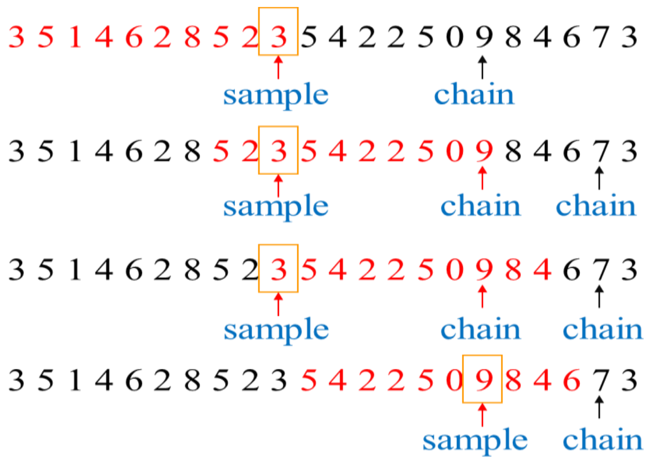


Figure 9: Chain Sampling example

The chain sampling algorithm works as follows:

- Include the *i*-th element that arrives in the sample with probability *1/min(i,n)*, that is, the previously sampled element is not discarded with probability *1- 1/min(i,n)*. The previously sampled element that is discarded cascades its "chain" as well,
- As each element is added to the sample, choose its "chain" as the index of the element that will replace it when it expires. When the *i*-th element expires, the window will include indices in the range $i+1 \ldots i+n$, so choose the index from this range uniformly at random,
- Once the element with that index arrives, store it and choose the index that will replace it in turn, building a "chain" of potential replacements

Chain sampling yields an expected memory usage of $O(k)$ which turns to $O(k\log(n))$ with high probability.

2.2.9 GK Quantiles

The GK algorithm [27] maintains a quantile summary Q as a collection of s tuples $t_0, t_2, \ldots, t_{s-1}$ where each tuple t_i is a triplet (v_i, g_i, Δ_i): (i) a value v_i that is an element of the ordered version of the incoming stream (set) S (ii) the value g_i is equal to $rmin_{GK}(v_i) - rmin_{GK}(v_{i-1})$ (for $i = 0$, $g_i = 0$) and (iii) the value Δ_i which equals $rmax_{GK}(v_i) - rmin_{GK}(v_i)$. In general, $rmin(v)$ corresponds to a lower bound on the rank of v (whatever is included in the parenthesis), while $rmax(v)$ is an upper bound on the rank of v in S. The elements $v_0, v_1, \ldots, v_{s-1}$ are in ascending order, v_0 is the minimum element in S and v_{s-1} is the maximum element in S. Note that $n = \sum_{j=1}^{s-1} g_j$ which is the number of elements seen so far in the stream and $rmin_{GK}(v_i) = \sum_{j\le i} g_j$, $rmax_{GK}(v_i) = \Delta_i + \sum_{j\le i} g_j$.

When a new element v arrives, first we search over the elements in Q to find an i such that $v_i < v < v_{i+1}$. A new tuple $t = (v, 1, \Delta)$ with $\Delta = \lfloor 2\varepsilon n \rfloor - 1$, for given ε-error parameter, is added to the summary where t becomes the new $(i + 1)$-st tuple. But the first $1/(2\varepsilon)$ elements are inserted in the summary with $\Delta_i=0$.

Then a merging operation follows we view the number of elements that have been observed over the incoming stream as an indication of time. Suppose that v in $t = (v, 1, \Delta)$ arrives at n' and is placed in Q according to the above procedure with $\Delta \approx 2\varepsilon n'$. At $n>n'$, we term the capacity of the tuple t as $2\varepsilon n - \Delta$. As n is becoming higher with the arrival of new stream elements, the capacity of the tuple t increases as we can have more error (Δ remains steady, but $2\varepsilon n$ strictly increases with n). We can thus merge tuples t_l, t_{l+1},…, t_i into a single tuple t_{i+1} at (time) n with precision $\sum_{j=l}^{i+1} g_j + \Delta_{i+1} \le 2\varepsilon n$, $g_{i+1} = \sum_{j=l}^{i+1} g_j$.

The above described summary Q can be used to answer quantile queries with an εn additive error, at the time when n elements have been observed. In order to answer a query for any rank ρ, the algorithm first finds the index i such that $\rho - rmin_{GK}(v_i) \le \varepsilon n$ and $rmax_{GK}(v_i) - \rho \le \varepsilon n$. The answer to the query is then v_i. The maintenance of the summary requires $O((1/\varepsilon) \log(\varepsilon n))$ space.

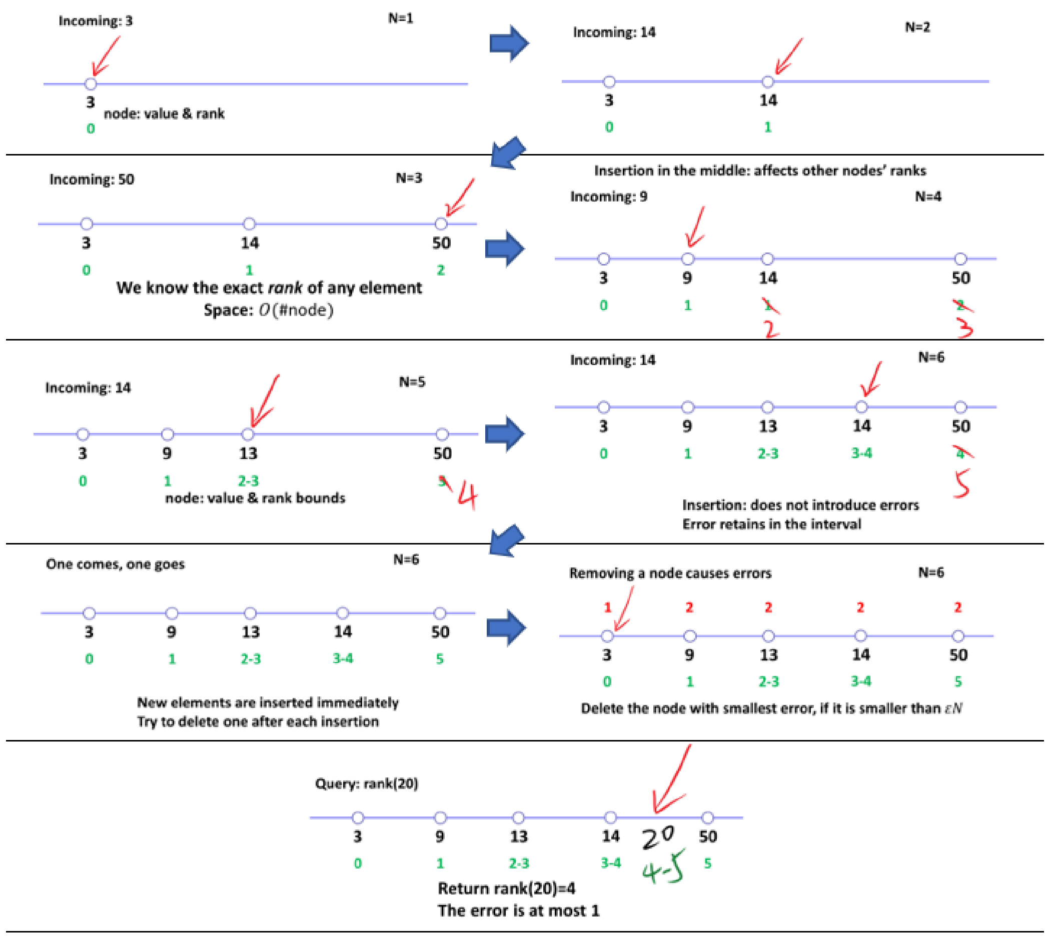

Incoming: 3
N=1
3
0
node: value & rank
Incoming: 14
N=2
3
14
0
1
Incoming: 50
N=3
3
14
50
0
1
2
We know the exact *rank* of any element
Space: O(#node)
Insertion in the middle: affects other nodes' ranks
Incoming: 9
N=4
3
9
14
50
0
1
~~1~~ 2
~~2~~ 3
Incoming: 14
N=5
3
9
13
50
0
1
2-3
~~3~~ 4
node: value & rank bounds
Incoming: 14
N=6
3
9
13
14
50
0
1
2-3
3-4
~~4~~ 5
Insertion: does not introduce errors
Error retains in the interval
One comes, one goes
N=6
3
9
13
14
50
0
1
2-3
3-4
5
New elements are inserted immediately
Try to delete one after each insertion
Removing a node causes errors
N=6
1
2
2
2
2
3
9
13
14
50
0
1
2-3
3-4
5
Delete the node with smallest error, if it is smaller than εN
Query: rank(20)
3
9
13
14
20
50
0
1
2-3
3-4
4-5
5
Return rank(20)=4
The error is at most 1

Figure 10: GK Quantile – Progressive insert and merge, query operation

2.3 Distributed Frameworks

Traditional database system cannot manage the always increasing volume and velocity of nowadays data. For this reason, the last two decades a variety of distributed frameworks have been built that can manage current requirements. These frameworks can utilize a network of computers (a cluster) and by allocating distributed computational power and storage are solving problems involving massive amounts of data. Also, they take measures for hardware failures and provide event delivery guarantees that are commonly addressed problems in the distributed system field. Although the SDE component could have been implemented in any modern Big Data platform that supports stream processing, such as Apache Spark, Flink, Storm, Kafka Streams and Akka, we decided to develop the SDE based on Apache Flink. The distributed framework Apache Flink with the combination of Apache Kafka as its input and output source can provide all the main requirements that would be discussed in chapter 3 for managing extreme scale interactive analytics in stream processing fashion with fault-tolerance and exactly once guarantees.

2.3.1 The Processing Framework: Apache Flink

Apache Flink is a distributed processing engine that is built from the bottom up to support data streaming processing in a stateful fashion. In general, Flink implements a data flow model in which data flows continuously though a network of transformation entities, creating a directed graph, most of the time a DAG(directed acyclic graph) that can be executed in a single pipeline or in a distributed parallel fashion.

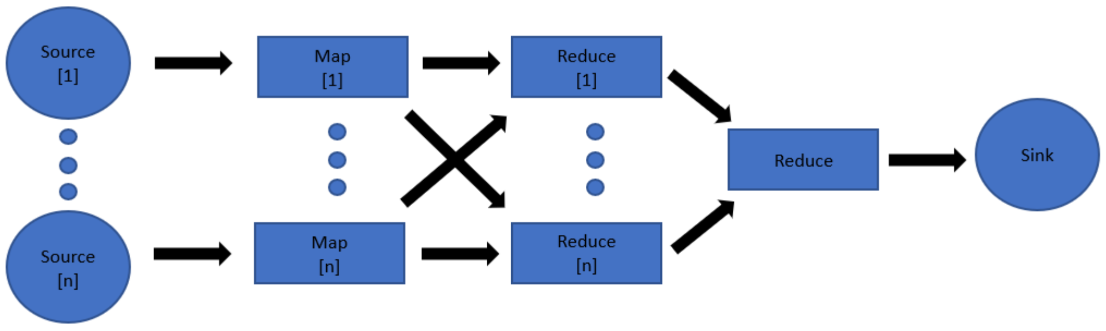


Figure 11: Example of a Flink Map Reduce Dataflow

This Dataflow is then sent to a Flink cluster to be executed. A Flink cluster is composed of (at least one) Master and a number of Worker nodes. The Master node runs a JobManager for distributed execution and coordination purposes, while each Worker node incorporates a TaskManager which undertakes the physical execution of tasks. Each Worker (JVM process) has a number of task slots (at least one). Each Flink operator may run in a number of instances, executing the same code, but on different data partitions. Each such instance of a Flink operator is assigned to a slot and tasks of the same slot have access to isolated memory shared only among tasks of that slot. The use of Dataflow management is common among most big data platforms but Flink also gives with special focus on time and state in an efficient and easy way. For data stream management is provides the user with the DataStream API. The API supported the creation of data streams from a variety of sources (Kafka, Cassandra, Elasticsearch, etc) then a series of transformation can be applied (below a small overview of the transformation that are used in this thesis is provided), and the final streams can be added to one or more output sources (sinks).

- **FlatMap**: Takes one element and produces one element.
- **KeyBy**: Logically partitions a stream into disjoint partitions. All records with the same key are assigned to the same partition
- **Connect**: "Connects" two data streams retaining their types. Connect allowing for shared state between the two streams.
- **CoFlatMap**: Similar with the FlatMap, it applied two FlatMap transformation one in each stream.
- **Union**: Union of two or more data streams creating a new stream containing all the elements from all the streams.
- **Split**: Split the stream into two or more streams according to some criterion.
- **Select**: Select one or more streams from a split stream.

This transformation are predefined entities that the user of the framework can extend and apply his own logic.

Another concept of that is essential for data stream processing is the State of the system. Flink manage state in a reliable way, that provides fault-tolerance by storing state at checkpoints and replaying the stream if failure occurs. In Flink every transformation has its own state, that be stored and accessed from many different places including memory, local files, or external key-value databases. Also, Flink exposes its managed state by a feature called Queryable state that allows user to query a job's state from outside. The Final property what also distinct Flink from most distributed streaming platforms is its notion of time. Flink given the user the ability to manage the:

- **Processing time**: Processing time refers to the system time of the machine that is executing the respective operation
- **Event time**: Event time is the time that each individual event occurred on its producing device, this can be achieved be out of order data. Event-time based application are important, however, unless the events are known to arrive in-order (by timestamp), event time processing incurs some latency while waiting for out-of-order events.
- **Ingestion time**: Ingestion time is the time that events enter Flink.

Other Desirable Features

- Flink provides a rich set of operators including native support for iteration
- With respect to memory management, Flink manages its own memory never breaking the JVM heap.
- Flink constitutes a Big Data platform of European origin and although lately acquired by Alibaba, recent H2020 projects [5] have been built on and have extended Flink's functionality. SDE component thus follows this successful paradigm.
- Flink exposes a metric system and can be connected to well-known reporters (Graphite, JMX, InfluxDB etc) or by provides the RestAPI that allows gathering and exposing metrics like memory and cpu usage, records processed, duration and many more.

2.3.2 The Messaging System: Apache Kafka

A Kafka cluster is composed of a number of brokers, run in parallel, that handle separate partitions of topics. Topics constitute categories of key-value messages where producers and consumers can write and read, respectively. Within a partition, messages are strictly ordered by their offsets (the position of a message within a partition) and

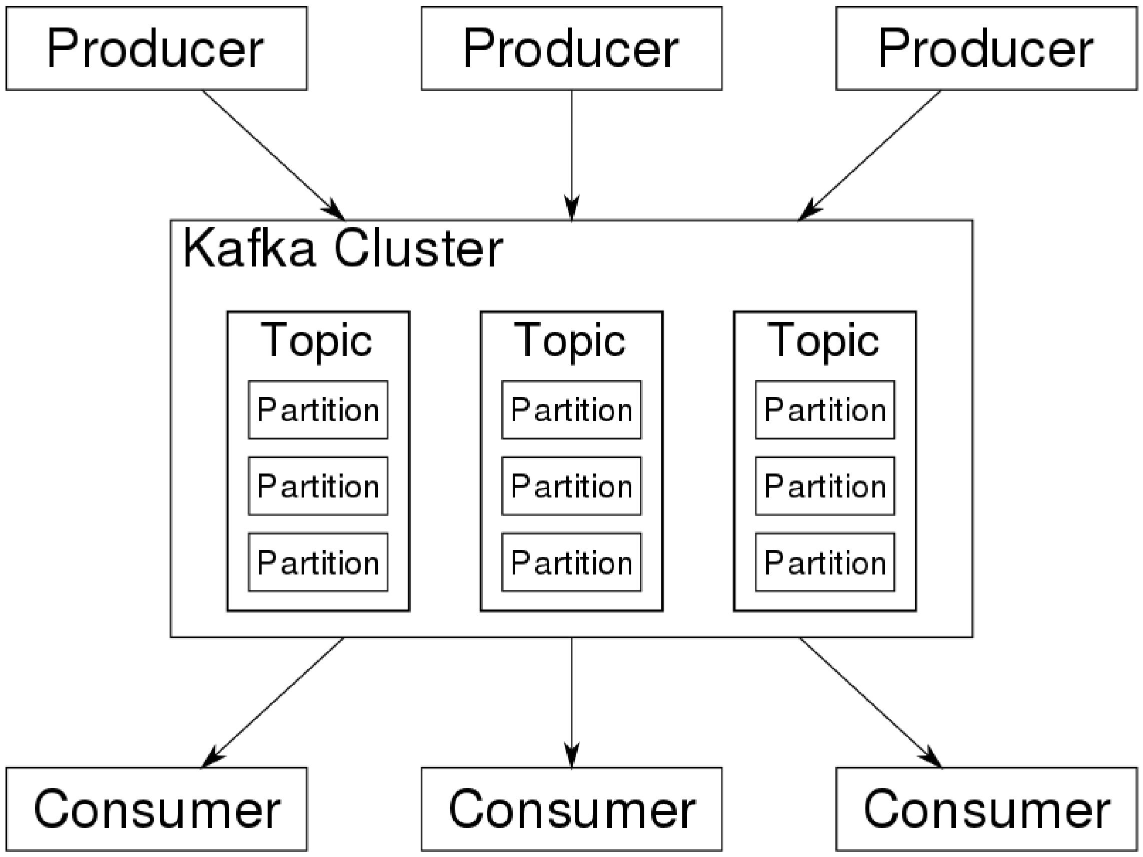


Figure 12: Kafka Overall Architecture

indexed and stored together with a timestamp. It also supports multiple reads and writes in each topic. Each producer and consumer have a GroupId, if two consumers with the same GroupId read from the same topic they will read different partitions, but if they belong to a different group both will read all the partitions. Kafka can work with all popular Big Data platforms and other storage or messaging engines including Flume, Spark, Storm and Flink for real-time ingestion, analysis, and processing of streaming data. Additionally, Kafka brokers support massive message streams for low-latency analysis in platforms supporting batch or hybrid workflows such as Hadoop or Spark, respectively. Kafka's popularity continuously grows and is nowadays a ubiquitous solution for scalable injection of Big Data.

3 Design of the Synopsis Data Engine

In this section we detail the functional and non-functional requirements that were taken into consideration during the design of SDE. In a nutshell, non-functional requirements define fundamental properties of a system and its components, essentially describing how a system should behave. On the other hand, functional requirements describe the actual functionality that the SDE provides as a component to a System or as a System itself.

The focus of this thesis is, by design, on the real-time, online analysis of massive streaming data. It can be applied to analyse streams of data, in many realistic scenarios like depicting the effect of drug combinations on cancer evolution simulations in real-time to stop the execution of unpromising simulations and free system resources for starting new ones or in Financial use case we need to monitor the behaviour of or detect correlations among stocks to identify investment opportunities and support predictions on system risks, respectively. Moreover, in a Maritime use case can perform real-time anomaly detection. To support these use cases and broader application scenarios was the rationale behind closing these requirements.

3.1 Non-functional Requirements

The SDE is design to support multiple use cases for different scenarios in an online fashion. Also, it should behave as service and as a stand-alone System and be a component to a more complex architecture. This creates a lot of restrictions on the way the system should behave. In the following sections we detail these restrictions.

3.1.1 Support Kappa Architectures

The Kappa architecture finds its applications in real-time processing of distinct events. Rather than using a relational Database like PostgreSQL, MySQL or even a key-value store like Cassandra, the data input in a Kappa Architecture system is a stream of continuous data. Data is streamed through a computational system for processing and state updates. Here is a basic diagram for the Kappa architecture:

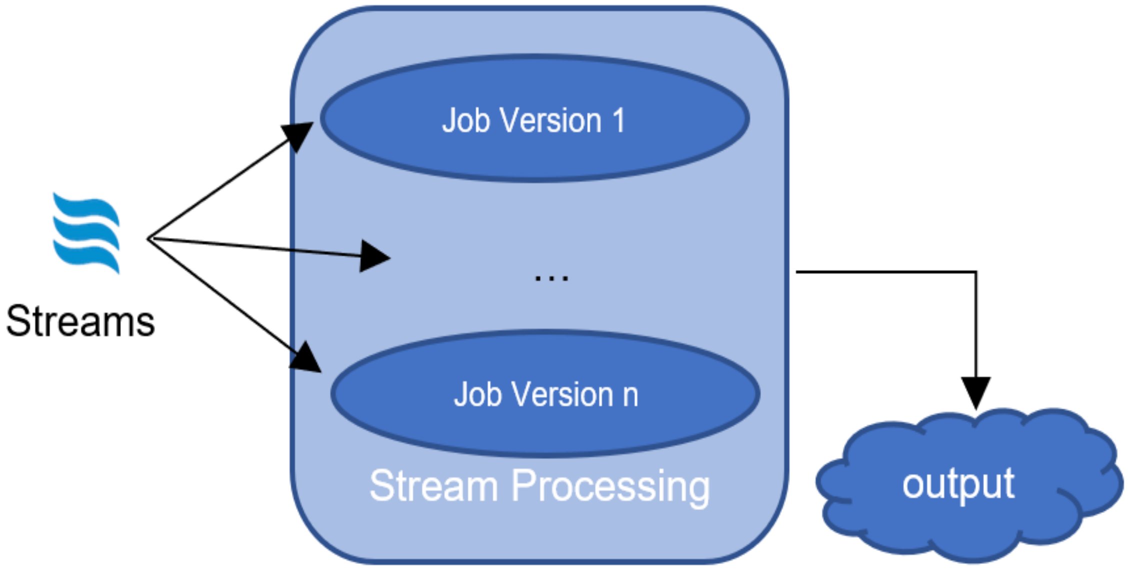


Figure 13: The Kappa Architecture

It processes data streams in real time and without the requirements of fixups or completeness, aims to minimize latency by using online algorithms on the most recent data. The output may not be as accurate or complete, but they are available almost immediately after data is received. Kappa architecture can be applied to develop data systems that are restricted to only (i)online algorithms and therefore

don't need batch processing, (iii) re-processing isn't required. The main advantages of using the Kappa architecture:

- Uses fixed memory
- Provides horizontally scalable systems
- Fewer resources are required as the machine learning is being done on the real time basis

3.1.2 Scalability Considerations

Scalability in the scope of this thesis comes in the following dimensions: (a) horizontal scalability, i.e., scaling with the volume and velocity of Big streaming Data, (b) vertical scalability, i.e., scaling with the number of processed streams, (c) federated scalability, i.e., the ability to scale the computation in settings composed of multiple, potentially geographically dispersed clusters.

State-of-the-art benchmarks on streaming Big Data platforms [29] report that Flink has a better overall throughput both for aggregation and join queries when compared with Spark and Storm. This shows that Flink either being viewed as a pure streaming engine (upon compared against Storm) or to support hybrid workflows (upon compared against Spark) provides increased horizontal scalability.

With respect to vertical scalability, the typical assumption of all existing Big Data platforms is that, while the data volume and the velocity of the data can be large, the number of distinct streaming sources is typically modest or small; thus, the focus is on techniques that can scale horizontally rather than vertically. For instance, the study of [29] is based only on a couple of streams. However, vertical scalability can find itself quite needed in broader application scenarios. As a concrete example, consider the problem of tracking the highly correlated pairs of stock data streams (under various statistical measures, e.g., Pearson correlation) over N distinct, high speed data streams, where N is a very large number. While several streaming techniques (e.g., based on synopses) are known for tracking the correlation of a given pair in space/time that is sublinear in the size of the streams, applying these ideas to track the full Θ(N2) correlation matrix results in a quadratic explosion in space and computational complexity which is simply infeasible for very large N. The problem is further

exacerbated when considering higher-order statistics (e.g., conditional dependencies/correlations). Clearly, techniques that can provide vertical scaling are sorely needed for financial use case scenarios. Therefore, our aim is to confront vertical scalability issues. To achieve that, we implement synopsis that not only parallelizes the processing among several workers in Flink, but also prunes unnecessary comparisons of stock streams that are known beforehand that cannot be correlated.

Finally, federated scalability is also quite often overlooked, cause almost all organization centralize their data in a single cluster/supercomputer. Despite that these organization often gather data from geographically dispersed sites. The implementation takes into consideration in the SDE architectural design as it is described in Chapter 4 in way that an organization can deploy SDEs in each site and gather a distributed answer minimizing the need for network bandwidth, cause only the answer travels though the network and not the whole data stream.

3.1.3 Pluggability Considerations

There are two dimensions of pluggability that are relevant to the SDE. First, a synopsis provided by the SDE is also destined to be used as an operator of a designed workflow. Each workflow, in turn, may engage streaming operators available in different Big Data platform implementations. We should thus ensure the pluggability of SDE's summarization operators to such generic workflows, ensuring that the SDE will be able to communicate with a variety of platforms. In order to abide by this non-functional requirement, SDE was built as a service so external source can query it and be SDE's input and output is provided via Apache Kafka.

Existing benchmarks [41] show that Kafka can serve the highest rates of data arrivals compared to other alternatives such as Apache ActiveMQ, a popular open-source implementation of JMS, and RabbitMQ, a message system known for its performance. Kafka is often used in real-time streaming data architectures to provide real-time analytics. Since Kafka is a fast, scalable, durable, and fault-tolerant publish-subscribe messaging system, it is used in use cases where JMS, RabbitMQ, and ActiveMQ may not even be considered due to volume and responsiveness issues.

The second level of pluggability required by many scenarios and ensured by Kafka involves the ability to work on top of many, potentially geographically dispersed clusters serving as a messaging cable connecting the entire federation.

3.1.4 Extensibility

The SDE is a system for different purposes and can be used be all kinds of user, so it should have the ability to extend and the level of effort that is required should be minimum. Extensions can be through the addition of new functionality or through modification of existing functionality. This is manly first be allowing the user to parametrize his synopsis and choosing which fields of a tuple he is going to use, so it supports different datasets.

Second, Flink Programming APIs include Java and Scala which support subtype polymorphism useful for ensuring pluggability of new synopsis in SDE’s libraries. In a nutshell, due to subtype polymorphism we allow the definition of an abstract synopsis class, which is refined by the implementation of synopsis-specific classes, one for each data summarization technique included in SDE’s libraries. Our current implementation uses Java for this purpose.

3.1.5 Reliability

Finally, but of equal importance is the reliability of our System we can break down to available and fault tolerant. Availability is the proportion of time a system is in a functioning condition. As it shown in Chapter 4, implementation decisions have been made to ensure the availability of our system by minimizing the query time, and in Chapter 6 with our experimentation we present how well our system is doing under extreme pressure. Fault tolerance is the property that enables a system to continue operating properly in the event of the failure. By carefully utilizing the properties of Flink State API (2.3.1.) so it doesn’t create an observable overhead, we can insure that the system will recover and presence of failure, and will eventually reach its desirable state.

3.2 Functional Requirements

Having discussed the non-functional requirements that led us to the adoption of Flink and Kafka for our SDE implementation, we proceed with the definition of functional requirements. The functional requirements involve the functionality the SDE needs to provide to external users, other Systems or as a part of a workflow. That is, given a workflow where a data summarization technique is included as a data synopsis operator, the implementation of that technique within the SDE receives input from upstream operators and may provide output streams to downstream operators of the workflow. Furthermore, the SDE exposes an application programming interface (API), that allow the system to be queried by the external systems or users regarding the available, currently maintained synopses. The SDE is equipped with an internal library of currently available synopses. The current status of the library is discussed in Section 4.6. According to the above observations the upper level functions the SDE needs to provide are identified below.

3.2.1 Single-stream Synopsis Maintenance

The SDE should allow building a synopsis on data originating from a single stream. For instance, in a Financial use case we may require sampling data involving trades of a particular stock. Similarly, in a Maritime use case, we may wish to maintain a sample of a single vessel's trajectory stream. Moreover, in a Life Science use case where a simulation of a tumour of realistic size can produce an amount of data of 100GB/min, the output of a simulation composes a stream and a sample of a monitored quantity needs to be maintained over it.

3.2.2 Dataset Synopsis Maintenance

The SDE should allow building a synopsis on data originating from a dataset, potentially composed of several different streams. For instance, in a Financial use case we may require sampling stock data of the whole set of monitored stock exchanges. Likewise, in a Maritime use case, we may wish to maintain a sample of trajectory positions of a group of vessels within a geographic region.

3.2.3 Multi-stream Synopsis Maintenance

The SDE should allow building a synopsis on data originating from a dataset composed of a number of streams, in a per stream fashion. Consider for instance stock exchanges in a Financial use case. If we wish to maintain a sample for each out of thousands of monitored stocks in the Financial dataset, what was mentioned in Section 3.2.1 is not adequate, since it entails that thousands of requests for single-stream synopsis maintenance should be issued towards the SDE. Moreover, in the a Life Science use case where a number of differently parameterized simulations run in parallel, each producing voluminous, high speed streams, an individual simulation may be viewed as a stream and a separate sample of monitored quantities needs to be maintained for each. The current functional requirement essentially means that a synopsis should be built for each separate stream included in the Financial or the Life Science dataset via a single such request.

3.2.4 Ad-hoc Querying Capabilities

A currently maintained synopsis should be able to accept one-shot, ad-hoc queries and provide respective approximate answers to downstream operators or application interfaces.

3.2.5 Continuous Querying Capabilities

A currently maintained synopsis should be able to accept continuous queries and provide answers to downstream operators or application interfaces every time the approximated quantity is updated either due to incoming tuples that alter the maintained synopsis and/or under some windowing (time – or count–based) operation.

3.2.6 Support Different Merge Techniques

A query to the SDE may produce many local answers from a Synopsis that is built on dataset (Section 3.2.2) or a Multi-stream Synopsis (Section 3.2.3) these local answers need to be aggregated with an aggregation function of the users choice like sum, or, and or just output all the local answers etc. so it can produce one global answer.

3.2.7 Dynamic Build/Stop of Synopses

This functional requirement concerns creating/deleting a synopsis on-the-fly, as the SDE component is up and running. Since a number of synopses may be maintained within the scope of currently running, streaming workflows, a new workflow that requires the maintenance of a new synopsis should be able to create such a synopsis without needing to restart the SDE component, i.e., without preventing the execution of already running processing pipelines.

3.2.8 Providing SDE Status Report

This functional requirement expresses the need of querying the SDE component for providing a list of the currently maintained synopses and their parameters. Such functionality may be of particular utility to any external client, in order to acquire a list of currently running synopses and check whether it can include an existing on in his execution plans or build a new one and use it.

This functional requirement is further refined to the following operations:

- Report a list of currently maintained/running synopses per stream.
- Report of all running synopses of a given stream.
- Report of all running synopses of a specific value field of a given stream.
- All the above are to be provided in case synopses are maintained on a dataset in its entirety instead of a single stream (see Section 3.2.2).
- All the above should also provide the parameter values of the synopses included in the respective reports. For instance, in case two different workflows have been built and use the same kind of different parameterized synopsis, the corresponding reports should provide this information.

3.2.9 External Synopsis Load and Maintenance

As already mentioned at the beginning of the current section, the SDE is equipped with a library of offered, implemented data summarization techniques. The contents of the current version of this library will be discussed shortly. Besides what is included in the current version of the aforementioned library, the SDE should be able to allow the dynamic loading of synopses originating from external libraries, so as to cover all possible application scenarios where application workflows require domain specific synopses, i.e., beyond popular ones covering broad application scenarios.

4 Implementation

In this section we detail our implementation techniques starting with the SDE architectural components and present their utility in serving the functional requirements identified in Section 3.2, then we give some insides on the SDE library and closing with some insides on the application programming interface that lets external users to harness the SDE functionality. Before proceeding to explaining the functionality of each component and the way they cooperate in the scope of the SDE architecture, we discuss the fundamentals of the parallelization schemes utilized within the SDE.

4.1 Parallelization Scheme(s)

The parallelization scheme that is employed in the design of the SDE is key-based parallelization. That is, every data tuple that streams in the SDE architecture and is destined to be included in one or more maintained synopses, does so based on the key it is assigned to it. When a synopsis is maintained for a particular stream (see functional requirements in Section 3.2.1 and Section 3.2.3) the key that is assigned to the respective update (newly arrived data tuple) is the StreamID of that particular stream for which the synopsis is maintained. In this case, within the distributed computation framework of Flink, all those streams with the same StreamID are processed by the same worker instance and parallelization is achieved by distributing the number of StreamIDs for which a synopsis is built, to the available workers instances in the cluster hosting the SDE. On the other hand, when a synopsis involves a dataset in its entirety (see respective functional requirement in Section 3.2.2) the desired degree of parallelism is included as a parameter in the respective request to build/start maintaining the synopsis (see function requirement in Section 3.2.7). In the latter case, one dataset is partitioned to the available workers in a round-robin fashion and the respective keys are created by the SDE, as we describe in detail later on, each of which corresponds to a particular worker.

There are two important remarks that we need to make at this point. The first one links the key assignment in the case of Dataset Synopsis Maintenance (Section 3.2.2) with the nature of the maintained synopsis. In order to distribute the load among the available workers in the way described above, the synopsis itself needs to possess the Composability property (Section 2.2). Composability, refers to the ability of building synopsis on parts of the data and then having a way to merge the partial synopses into one synopsis, which will be equivalent to the synopsis that would have been built in case the synopsis was maintained centrally (instead of distributivity). For instance, in case FM sketches or Bloom Filters [17] (bitmaps) are built on different data partitions at separate worker nodes, they can be merged into one FM sketch or Bloom Filter via simple logical disjunction (OR) or conjunction (AND) operations. In case a maintained synopsis does not possess the mergeability property, the corresponding synopsis maintenance request should be parameterized with a unitary parallelization degree. The second remark we need to make here is that in the current design of the SDE, this is supposed to run on a number of worker nodes of a cluster, which poses an upper limit on the possible parallelization degree.

4.2 Data and Query Ingestion

Having clarified the above, we proceed with describing the flow of data and of the queries (requests) in the SDE architecture shown in Figure 14. Data and request streams arrive at a particular Kafka topic each. In the case of the DataTopic of Figure 14, a parser component is used in order to extract the value of the field(s) on which a currently running synopsis is maintained. The respective parser of the RequestTopic topic reads the request and processes it. When an incoming request involves the maintenance of a new synopsis, the parser component extracts information about synopsis parameters

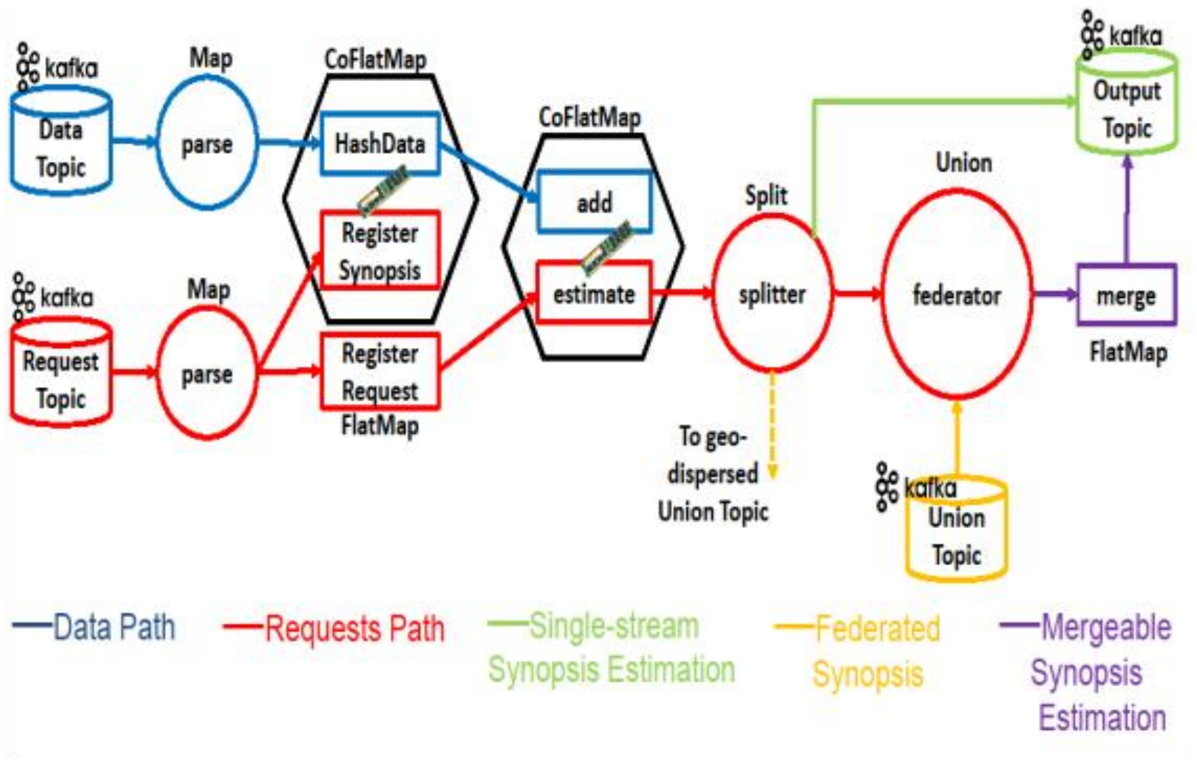


Figure 14: The SDE Architecture

(see for instance Figure 17) and nature, i.e., whether it is on a single stream, on a dataset, or involves a multi-stream synopsis maintenance request. In case the request is an ad-hoc query (see functional requirement in Section 3.2.4) the parser component extracts the corresponding synopsis identifier(s).

4.3 Requesting New Synopsis Maintenance

When a request is issued for maintaining a new synopsis, it initially follows the red-colored paths of the SDE architecture. That is, the corresponding parser sends the request to a FlatMap operator (termed RegisterRequest at the bottom of Figure 14) and to another FlatMap operator (RegisterSynopsis) which is part of a CoFlatMap one. In a nutshell, a FlatMap operator takes one tuple and produces zero, one, or more tuples, while a CoFlatMap operator hosts two FlatMaps that can share the state of common variables (therefore the linking icon in the figure) among streams that have previously been connected (using a Connect operator in Flink). RegisterRequest and RegisterSynopsis produce the keys as analysed in Section 4.1 for the maintained synopsis but provide different functionality. The RegisterRequest operator uses these keys in order to later decide which worker(s) an ad-hoc query, which also follows the red-coloured path, as explained shortly, should reach. On the other hand, the RegisterSynopsis operator uses the same keys to decide to which worker(s) a data tuple destined to update one or more synopses should be directed. Such an update follows the blue-coloured path in Figure 14, The possible parallelization degree of the corresponding RegisterSynopsis and RegisterRequest operators, beyond being affected by the number of available worker nodes, it is restricted by the number of maintained synopses.

4.4 Updating the Synopsis

When a data tuple destined to update one or more synopses is injected via the DataTopic of Kafka it follows the blue-coloured path of the SDE architecture. The tuple is directed to the HashData FlatMap of the corresponding CoFlatMap where the keys (StreamID for single stream synopsis and/or WorkerID for Dataset Synopsis) are looked up based on what RegisterSynopsis has created. Following the blue-colored path, the tuple is directed to an add FlatMap operator which is part of another CoFlatMap. The add operator updates the maintained synopsis as prescribed by the algorithm of the corresponding technique. For instance, in case a BloomFilter sketch is maintained, the add operation hashes the incoming tuple to a position of the maintained bitmap and turns the corresponding bit to 1 if it is not already set. Notice, that the blue-coloured path in Figure 14 remains totally detached from the red-coloured path. This depicts a design choice we follow for facilitating ad-hoc querying capabilities. That is, since the data updates on several maintained synopses may arrive at an extremely high rate, typically a lot higher than the rate at which ad-hoc queries are issued, in case of the two paths were crossing at some point of the architecture, back-pressure on the blue-coloured path would also stall the execution of ad-hoc queries. Having kept the two paths independent, data updates and ad-hoc queries are inserted in different processing queues and thus, even when the queue of the data updates grows, ad-hoc queries can be answered in a timely manner, based on the current status of the maintained synopses.

4.5 Ad-hoc Querying Answering

An ad-hoc query arrives via the RequestTopic of Kafka and is directed to the RegisterRequest operator. The operator, which produces the keys in the same way as RegisterSynopsis does, looks up the key(s) of the queried synopsis and directs the corresponding request to the estimate FlatMap operator of the corresponding CoFlatMap. The estimate operator reads via the shared state the current status of the maintained synopsis and extracts the estimation of the corresponding quantity the synopsis is destined to provide. For instance, upon performing an ad-hoc query on an BloomFilter sketch, the estimate operator reads the maintained bitmap, gets the Key from the query parameters and provides a membership estimation using the index of the Key answers true or false. Figure 17 provides a summary of the estimated quantities each of the currently supported synopsis can provide.

4.6 The SDE Library

The internal structure of the SDE library is a combination of a list of synopses and one of reduce functions. That Synopsis library is illustrated in Figure 15 which also provides only a partial view of the supported synopses for readability purposes. Figure 17 provides a full list of currently supported synopses, their utility in terms of approximation quantities and their parameters and in Figure 19 a list of supported reduce function. The development of the SDE library exploits subtype polymorphism in order to ensure the desired level of extensibility for new synopses and reduce functions definitions (see Section 3.1.4).

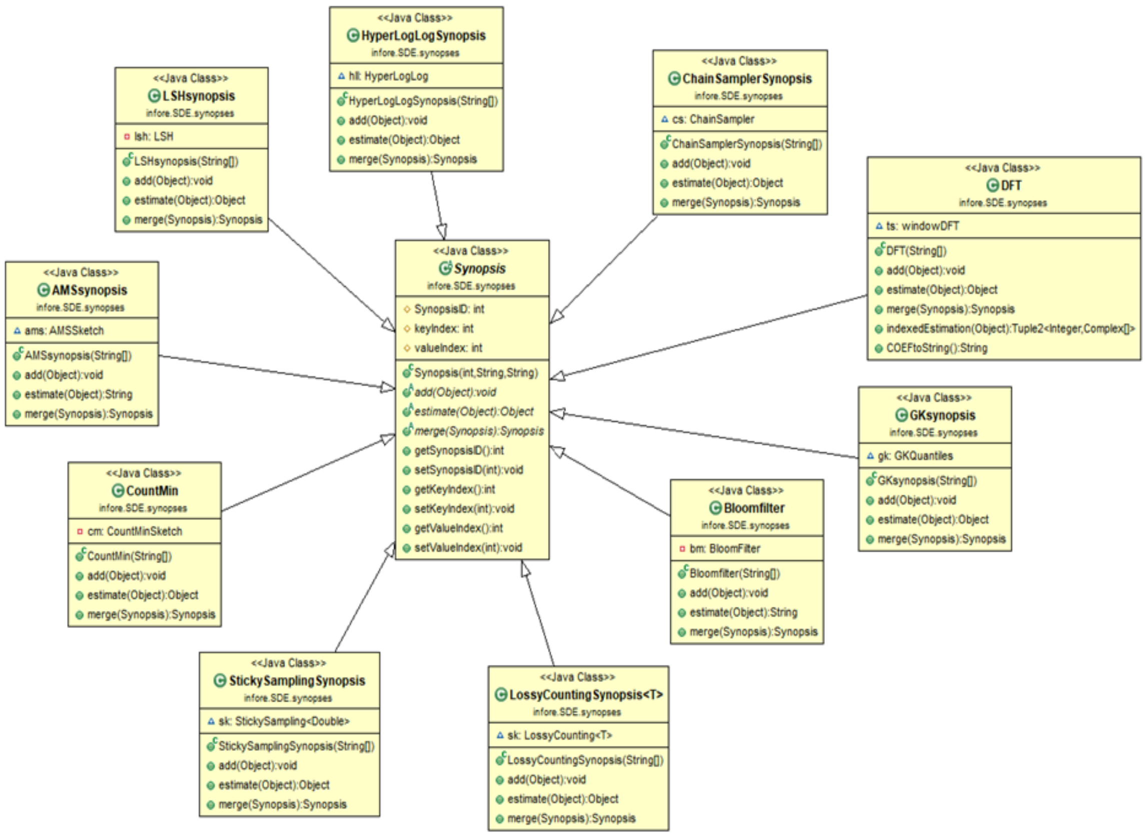


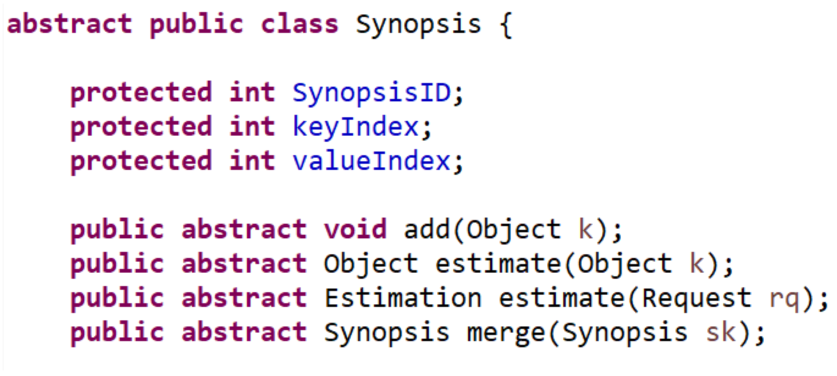
Figure 15: UML diagram of the Synopses Library

4.6.1 Synopsis library

As Figure 15 shows there is a higher-level class called Synopsis with attributes related to:

- A unique identifier that is essential for separating synopsis with each other for querying and deletion purposes.
- A String that includes information about the index of the key field in an incoming data tuple that refers to the field that the synopsis needs to use as a key. A lot of synopsis operates in a key-value fashion like (CountMin, AMS, etc.) The key field definition is not always mandatory.
- A respective index of the value field, i.e., the field which the summary is built on.
- The constructor of the class receives another String including parameters related to the identifier of a particular type of synopsis as well as synopsis specific parameters as included Figure 19.

- Furthermore, the Synopsis class includes methods for add, a data point to the synopsis (update), estimate, receives a request and outputs an estimation, and a merge method for combining synopsis.



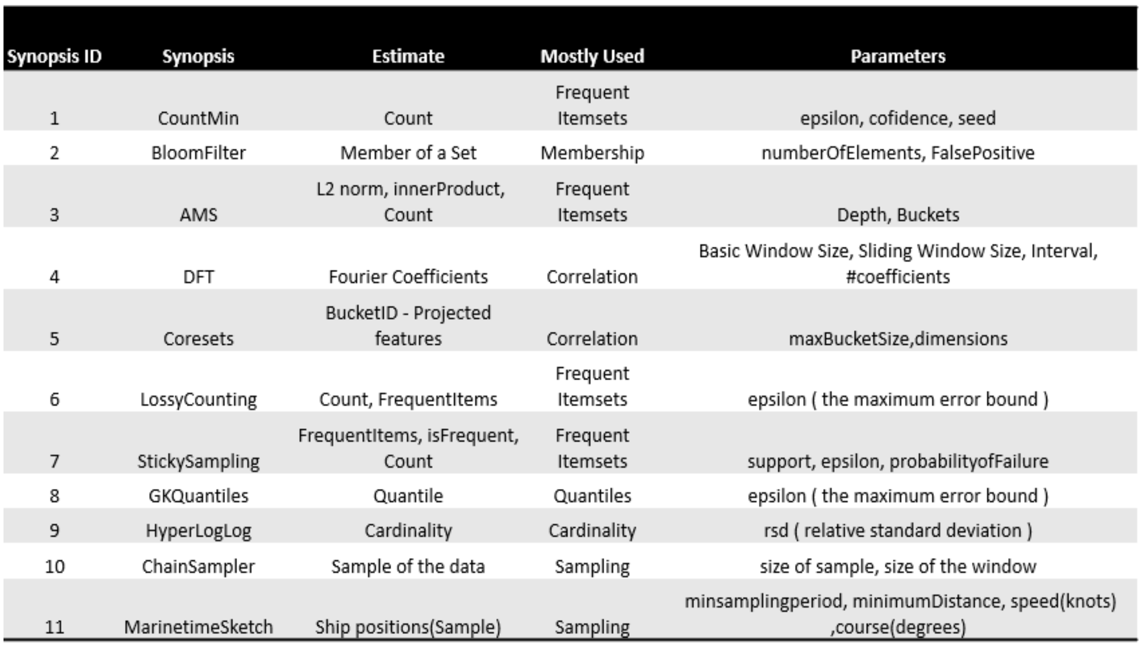
```
abstract public class Synopsis {

    protected int SynopsisID;
    protected int keyIndex;
    protected int valueIndex;

    public abstract void add(Object k);
    public abstract Object estimate(Object k);
    public abstract Estimation estimate(Request rq);
    public abstract Synopsis merge(Synopsis sk);
```

Figure 16: The abstract Synopsis Class

Every specific synopsis algorithm is implemented in a separate class, that extends Synopsis and overrides the add, estimate, and merge methods with the algorithmic details of that technique.

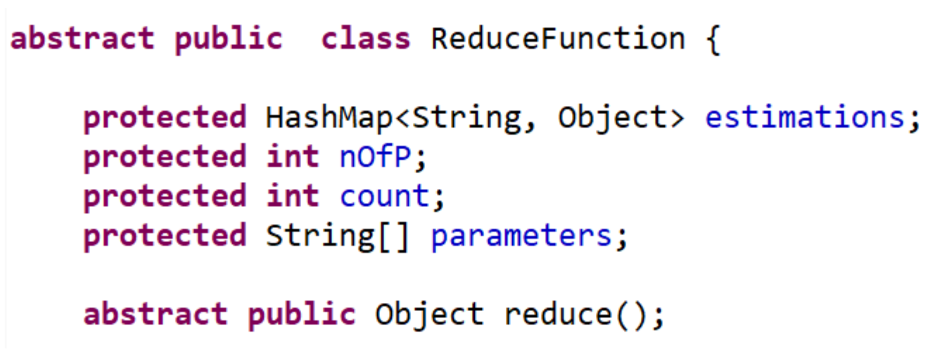


Synopsis ID	Synopsis	Estimate	Mostly Used	Parameters
1	CountMin	Count	Frequent Itemsets	epsilon, cofidence, seed
2	BloomFilter	Member of a Set	Membership	numberOfElements, FalsePositive
3	AMS	L2 norm, innerProduct, Count	Frequent Itemsets	Depth, Buckets
4	DFT	Fourier Coefficients	Correlation	Basic Window Size, Sliding Window Size, Interval, #coefficients
5	Coresets	BucketID - Projected features	Correlation	maxBucketSize,dimensions
6	LossyCounting	Count, FrequentItems	Frequent Itemsets	epsilon (the maximum error bound)
7	StickySampling	FrequentItems, isFrequent, Count	Frequent Itemsets	support, epsilon, probabilityofFailure
8	GKQuantiles	Quantile	Quantiles	epsilon (the maximum error bound)
9	HyperLogLog	Cardinality	Cardinality	rsd (relative standard deviation)
10	ChainSampler	Sample of the data	Sampling	size of sample, size of the window
11	MarinetimeSketch	Ship positions(Sample)	Sampling	minsamplingperiod, minimumDistance, speed(knots) ,course(degrees)

Figure 17: Supported Synopses in SDE

4.6.2 Supported Reduce Function

In the same notion as the Synopsis Library we support a variety of reduce function the reduce abstract class is show in the Figure 18 with attributes referring to:



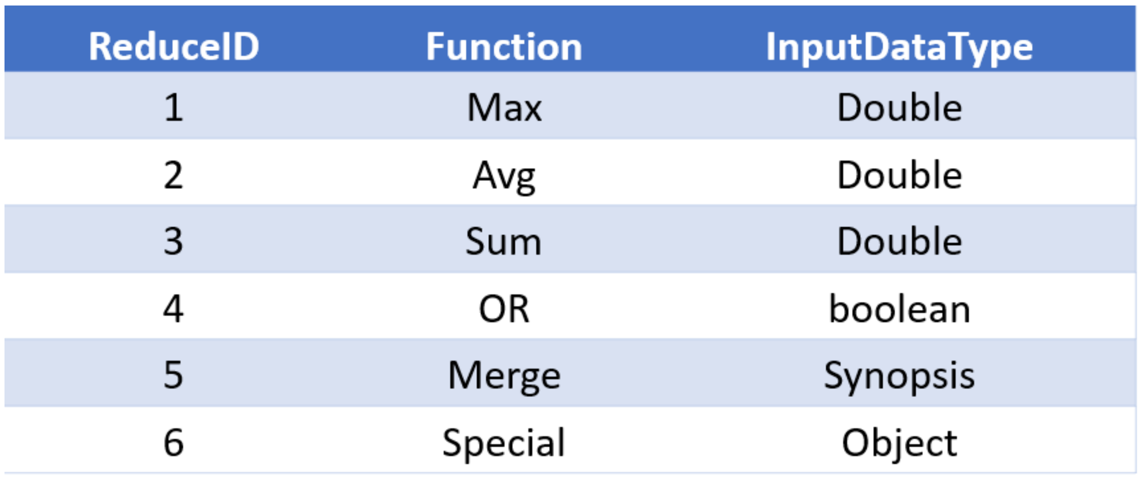
```java
abstract public  class ReduceFunction {

    protected HashMap<String, Object> estimations;
    protected int nOfP;
    protected int count;
    protected String[] parameters;

    abstract public Object reduce();
```

Figure 18: The Reduce Function abstract class

- ➢ A HashMap that keeps track of all the incoming estimation that have been received so far.
- ➢ The number of total estimations the function receives before starting the reduce phase.
- ➢ The number of received estimations
- ➢ The parameters of the received request
- ➢ A reduce function which by processing all the incoming estimation produces one estimation the final answer of that request (query).

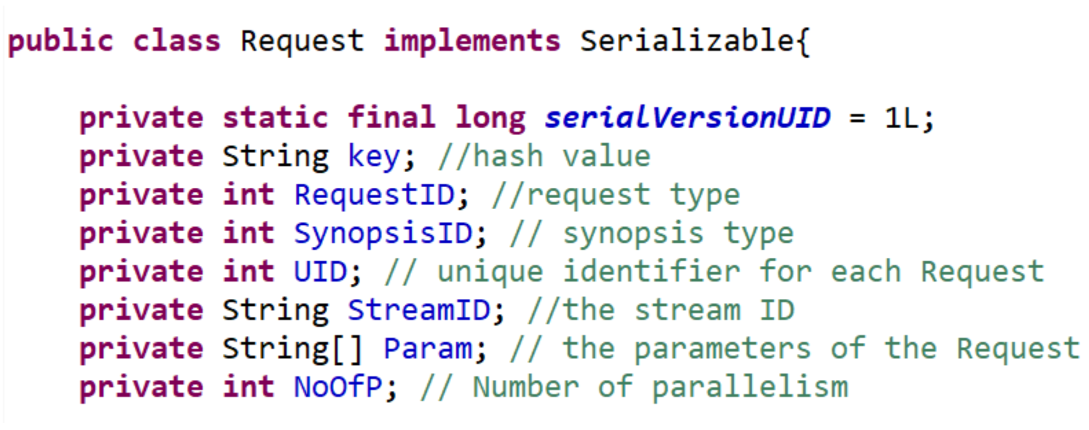


ReduceID	Function	InputDataType
1	Max	Double
2	Avg	Double
3	Sum	Double
4	OR	boolean
5	Merge	Synopsis
6	Special	Object

Figure 19: The list of Supported Reduce Functions

4.7 The Application programming interface (API)

This work includes a computing interface (API) that is built on top of Kafka messages that defines how other external users can use our system. It provides three calls that can be made, add, delete and estimate on a synopsis UID. These calls can be made by sending messages to the right kafkaTopic while following a specific format. That message should include the following fields as shown in Figure 20. Two String field, when requesting an add or delete synopsis the key field should be the dataset key if the user requires a Dataset Synopsis, or a Multi-stream Synopsis as shown in 3.2.1 and 3.2.2, or it can be same as the second String field the StreamID for the functionality of Single-stream Synopsis 3.2.3. A String Array that provides the synopsis parameters for adding new synopsis (see Figure 17 and the query parameters when querying a synopsis.



```java
public class Request implements Serializable{

    private static final long serialVersionUID = 1L;
    private String key; //hash value
    private int RequestID; //request type
    private int SynopsisID; // synopsis type
    private int UID; // unique identifier for each Request
    private String StreamID; //the stream ID
    private String[] Param; // the parameters of the Request
    private int NoOfP; // Number of parallelism
```

Figure 20: The Request Class

And Finally four Integer fields, the number of parallelism the synopsis have, a RequestID that defines the type of the request (add (1), delete(2), estimate(3)), the SynopsisID that defines the type of the synopsis (see Figure 17 and a unique identifier for each Synopsis that should be the same when adding, deleting or estimating on a specific synopsis.

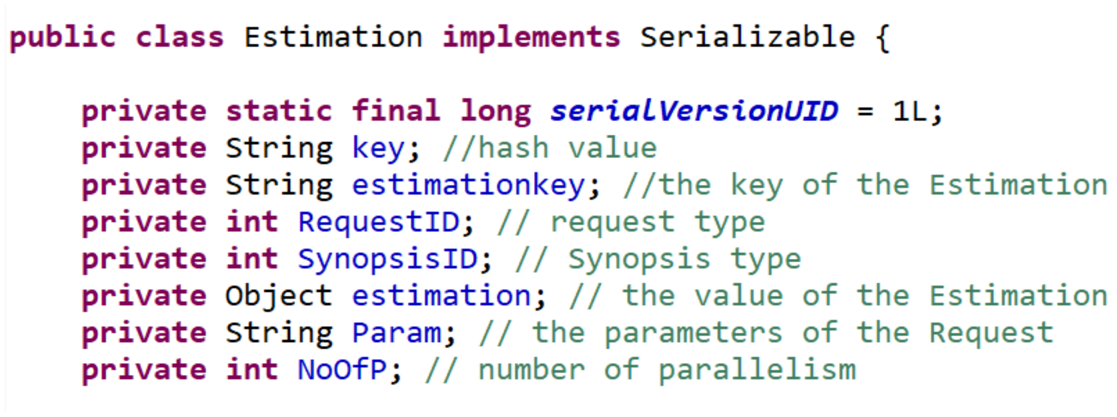
The format of the message is as follows:

```
Value<DatasetKey,RequestID, UID, SynopsisID, StreamID, parameters,
NumberOfParallelism>
```

Example for adding a CountMin synopsis for a financial data set

"FINANCIAL_USECASE,1, 111, 1, INTEL, 1;2;0.0002;0.99; 4,1"

Similar to the input calls the output of a query is provided by a Kafka message that has the fields shown in Figure 21. The only difference with the fields described above is the estimationkey and the estimation that are synopsis specific for example, in a countMin query the output is the estimated count and in a DFT the Fourier coefficients.



```
public class Estimation implements Serializable {

    private static final long serialVersionUID = 1L;
    private String key; //hash value
    private String estimationkey; //the key of the Estimation
    private int RequestID; // request type
    private int SynopsisID; // Synopsis type
    private Object estimation; // the value of the Estimation
    private String Param; // the parameters of the Request
    private int NoOfP; // number of parallelism
```

Figure 21: The Estimation Class

And it also follows a similar format

```
Value<estimationkey, StreamID, UID, RequestID, SynopsisID, estimation,
parameters, NumberOfParallelism>
```

Example of an estimation showing the answer of the above example query

```
<111, INTEL, 111, 3, 1, 38548, INTEL, 1 >
```

5 EXPERIMENTAL EVALUATION

5.1 Cluster setup up

To test the performance of our SDEaaS approach, we utilize a Kafka cluster with 3 Dell PowerEdge R320 Intel Xeon E5-2430 v2 2.50GHz machines with 32GB RAM each and one Dell PowerEdge R310 Quad Core Xeon X3440 2.53GHz machine with 16GB RAM and a Flink cluster has 10 Dell PowerEdge R300 Quad Core Xeon X3323 2.5GHz machines with 8GB RAM each. Both Kafka and Flink were provided by SoftNet lab at Technical University of Crete. Note that our experiments concentrate on computational and communication performance figures. We do not provide results for the synopses accuracy, since our SDEaaS approach does not alter in anyway the accuracy guarantees of synopses. Theoretic bounds and experimental results for the accuracy of each synopsis can be found in the reference.

5.1.1 Dataset

For dataset we use a real dataset composed of ∼50 different stock markets each on with different stocks as shown the Figure 22. In total a sum of ~5000 stocks contributing a total of ∼10 TB of Level 1 and Level 2 data provided to us by http://www.springtechno.com/ in the scope of the EU H2020 INFORE project (http://infore-project.eu/) acknowledged in this thesis. More precisely, Level 1 data involve stock trades of the form <Date, Time, Price, Volume > for each data tick of an asset (stock). Level 2 data show the activity that takes place before a trade is made. Such an activity includes information about offers of shares and corresponding prices as well as respective bids and prices per stock. Thus, Level 2 data are shaped like series of < Ask price, Ask volume, Bid price, Bid volume > until a trade is made. These pairs are timestamped by the time the stock trade happens. The higher the number of such pairs for a stock, the higher the popularity of the stock.

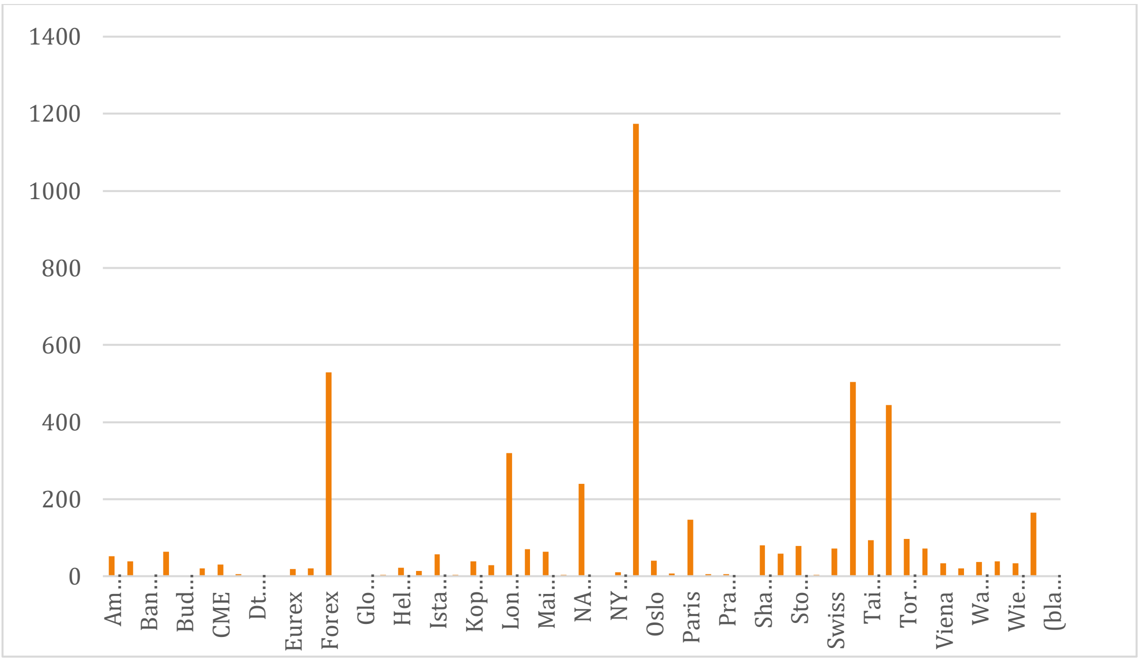


Figure 22: Number of Stocks per Stock Market in our Dataset

5.2 Assessing Scalability

In the experiments of this first set, we test the performance of our SDEaaS approach alone. That is, we purely measure its performance on maintaining various types of synopses. In particular, we measure the throughput, expressed as the number of tuples being processed per time unit (second) and communication cost (Giga bytes) among workers, while varying a number of parameters involving horizontal ((i),(ii)), vertical (iii) and federated (iv) scalability, respectively:

1) the parallelization degree [2-4-6-8-10]
2) the update ingestion rate [1-2-5-10] times the Kafka ingestion rate (i.e., each tuple read from Kafka is cloned [1-2-5-10] times in memory to further increase the tuples to process)
3) the number of summarized stocks (streams) [50-500-5000] and
4) the Giga bytes communicated among workers for maintaining each examined synopsis as a federated one. Note that this also represents the communication cost that would incur among equivalent number of sites (computer clusters), instead of workers, each of which maintains its own synopses.

In each experiment of this set, we build and maintain Discrete Fourier Transform (DFT–8 coefficients, 0.9 threshold), HyperLogLog (HLL – 64 bits, m = 3), CountMin (CM – $\epsilon = 0.002$, $\delta = 0.01$) synopses each of which, is destined to support different types of analytics related to correlation, cardinality and distinct count estimation, respectively (Table 1). All the above parameters were set after discussions with experts from the data provider and on the same ground, we use a time window of 5 minutes.

Figure 23 shows that increasing the number of Flink workers causes proportional increase in throughput. This comes as no surprise, since for steady ingestion rate and constant number of monitored streams, increasing the parallelization degree causes fewer streams to be processed per worker which in turn results in reduced processing load for each of them.

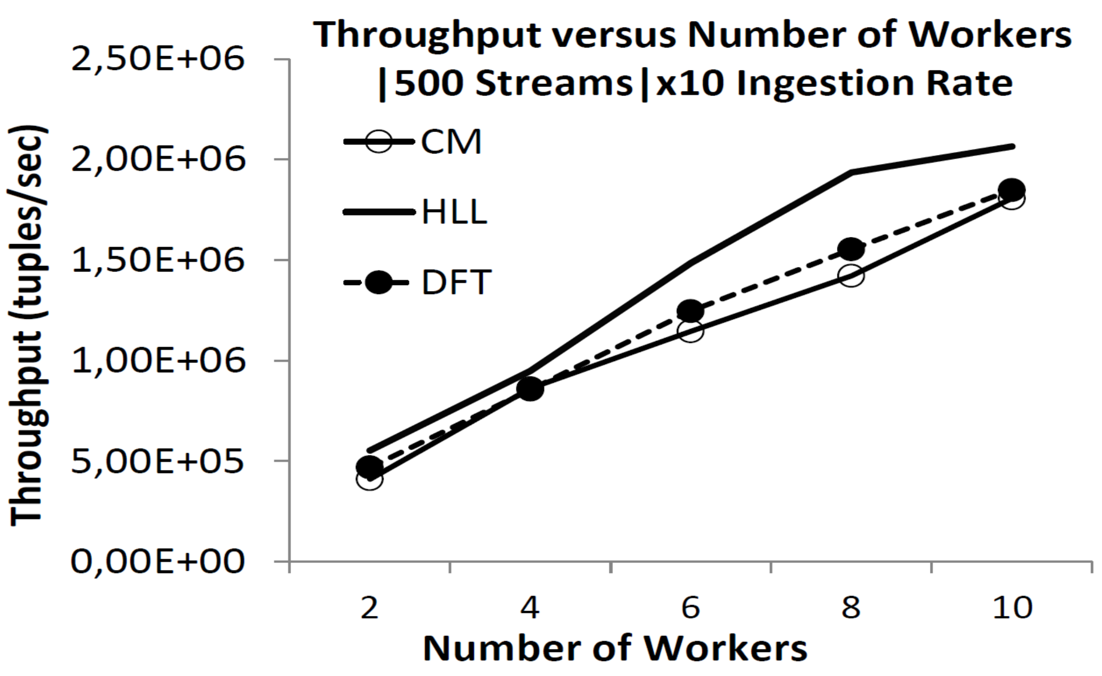


Figure 23 Throughput versus number of Workers

Figure 24, on the other hand, shows that varying the ingestion rate from 1 to 10 causes throughput to increase almost linearly as well. This is a key sign of horizontal scalability, since the figure essentially says that the data rates the SDEaaS can serve, quantified in terms of throughput, are equivalent to the increasing rates at which data arrive to it.

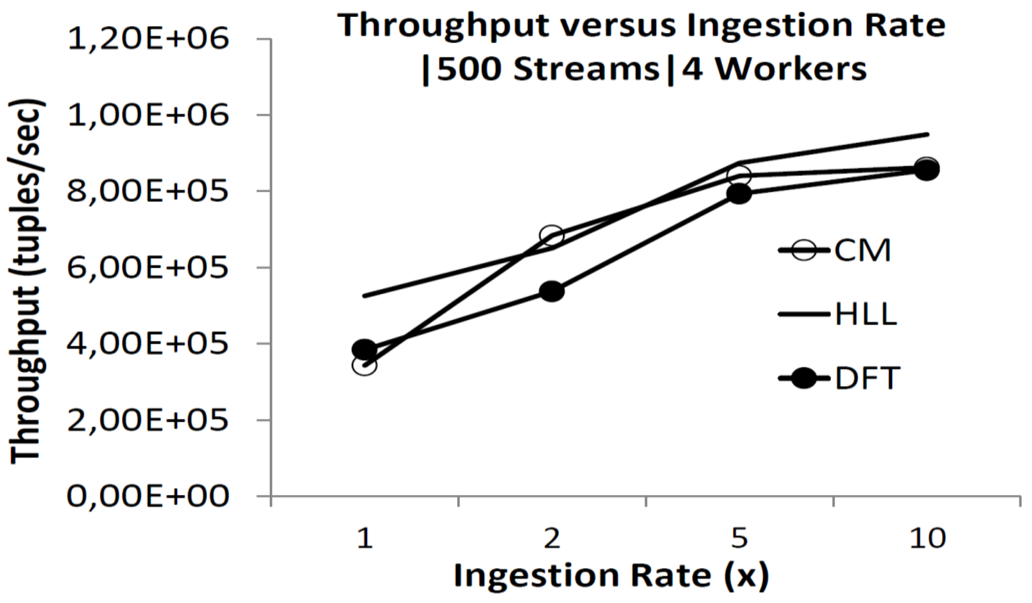


Figure 24: Throughput versus Ingestion Rate

Figure 25 shows something similar as the throughput increases upon increasing the number of processed streams from 50 to 5000. This validates our claim regarding the vertical scalability aspects the SDEaaS can bring in the workflows it participates.

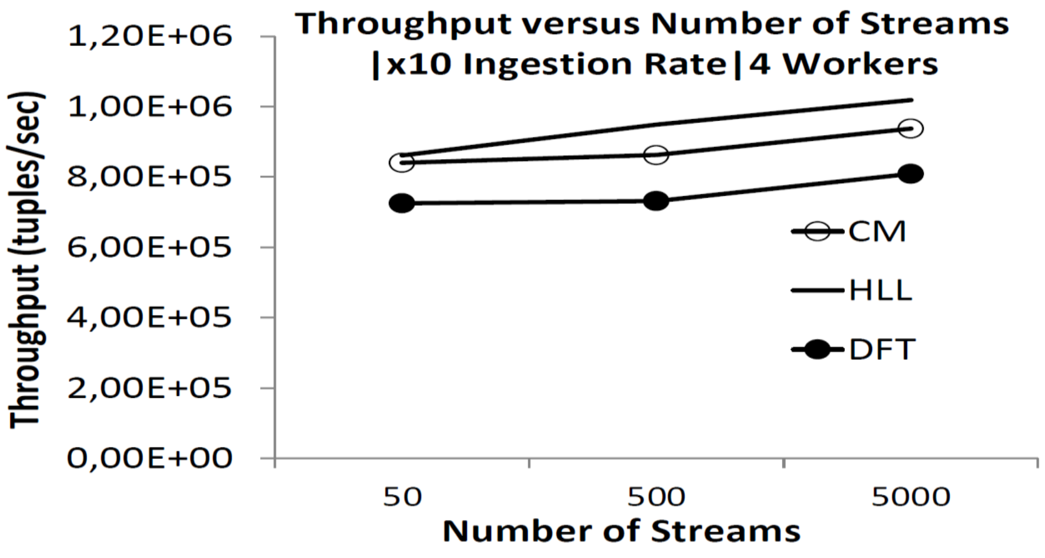


Figure 25: Throughput versus Number of Streams

Finally, Figure 26 illustrates the communication performance of SDEaaS upon maintaining federated synopses and communicating the results to a responsible site to derive the final estimations (see yellow arrows in Figure 14). For this experiment, we divide the streams among workers and each worker represents a site which analyses its own stocks by computing CM, HLL, DFT synopses. A random site is set responsible for merging partial, local summaries and for providing the overall estimation, while we measure the total Gbytes that are communicated among sites/workers as more sites along with their streams are taken into consideration.

Note that the sites do not communicate all the time, but upon an Ad-hoc Query request every 1 minute. Here, the total communication cost for deriving estimations from synopses, is not a number that says much on its own. It is expected of the communication cost will rise as more sites are added to the network. The important factor to judge federated scalability is the communication cost when we use the synopses ("CM+HLL+DFT") line in Figure 26 compared to when we do not. Therefore, in Figure 26, we also plot a line (labelled "NoCM+NoHLL+NoDFT") illustrating the communication cost that takes place upon answering the same (cardinality, count, time series) queries without synopses. As Figure 26 illustrates (the vertical axis is in log scale), the communication gains steadily remain above an order of magnitude.

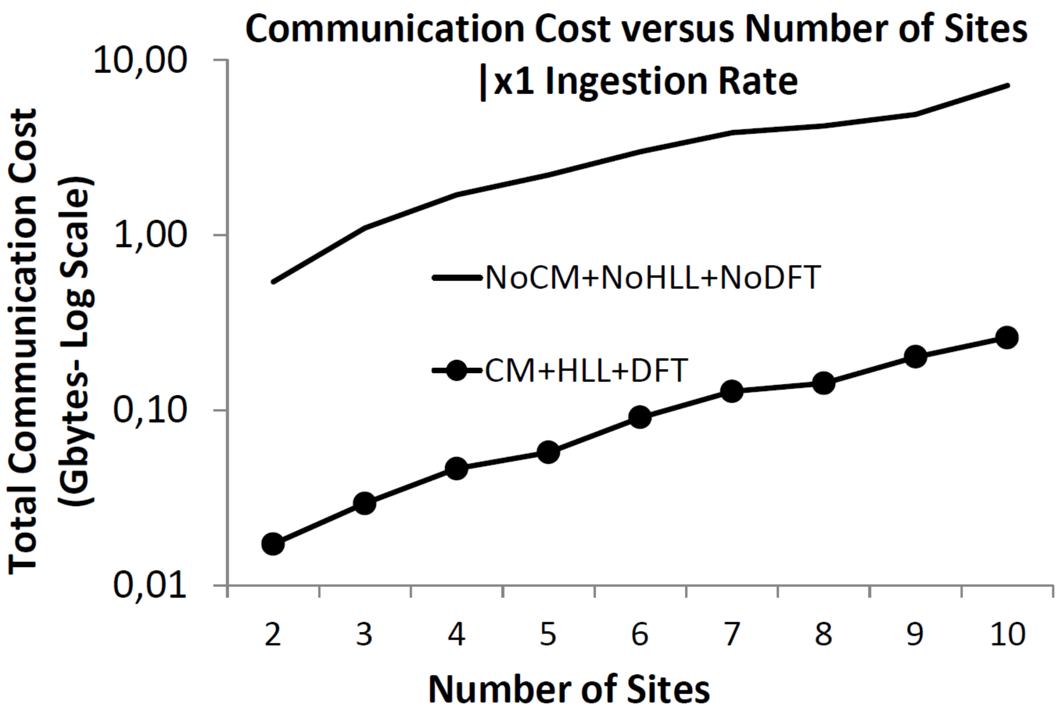


Figure 26: Communication Cost versus Number of Streams

5.3 Comparison against other Candidates

In the second group of experiments we choose a workflow that was inspired by a world-known benchmark [16] (the Yahoo! benchmark) which is a window, a Join operator followed by an aggregation.

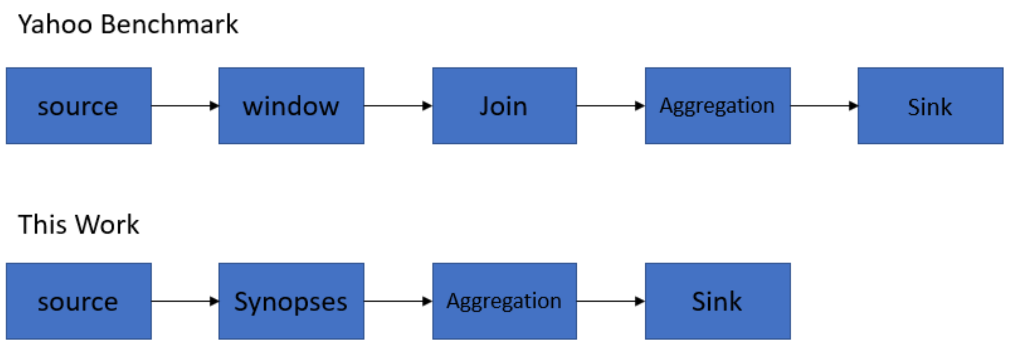


Figure 27: Changed Yahoo Benchmark

After a discussing with experts from the financial domain we conclude that applying the pair-wise correlation algorithm in the number of bids or each stock before a trade take place is a valuable knowledge. So, in our experiment workflow shown in Figure 27 we consider that both Level 1 and Level 2 data arrive at a Source. Then, we use a synopsis which first keeps counters per stock in each window (so the window operation is coded inside the synopsis). When a trade for a stock takes place, the corresponding Level 1 tuple is directed to the Synopsis that holds this stock's counters and the counter value updates a DFT synopsis. Then every 1 second (same as [16]) we aggregate all DFTs in a correlation function and forward all the correlation above a threshold.

In Figure 28 we measure the performance of our SDEaaS approach employed in this work against three alternative approaches. More precisely, the compared approaches are:

• Naive: This is the baseline approach which involves sequential processing of incoming tuples without parallelism or any synopsis.

• SDEaaS(DFT+Parallelism): This is the approach employed in this work which combines the virtues of parallel processing (using 4 workers in Figure 28) and stream summarization (DFT synopsis) towards delivering interactive analytics at extreme scale.

• Parallelism(NoDFT): This approach performs parallel processing (4 workers), but does not utilize any synopses to bucketize time series or reduce their dimensionality.

• DFT(NoParallelism): The DFT(NoParallelism) approach utilizes DFT synopses to bucketize time series and for dimensionality reduction, but no parallelism is used for executing the workflow. Pairwise similarity checks are restricted to adjacent buckets and thus comparisons can be pruned, but the computation of similarities is not performed in parallel for each bucket. This approach corresponds to competitors such as DataSketch [9] or Stream-lib [8] which provide a synopses library but do not include parallel implementations of the respective algorithms and do not follow an SDEaaS paradigm.

Each line in the plot of Figure 28 measures the ratio of throughputs of each examined approach over the Naive approach varying the amount of monitored stock streams. Let us first examine each line individually. It is clear that when we monitor few tens of stocks (50 in the figure), the use of DFT in the DFT(NoParallelism) marginally improves (1.5 times higher throughput) the throughput of the Naïve approach. On the other hand, the Parallelism(NoDFT) improves the Naive by ~2.5 times. Our SDEaaS(DFT+Parallelism), taking advantage of both the synopsis and parallelism improves the Naïve by almost 4 times. Note that when 50 streams are monitored, the number of performed pair-wise similarity checks in the workflow of Figure 27 for the Naive approach is 2.5K/2.

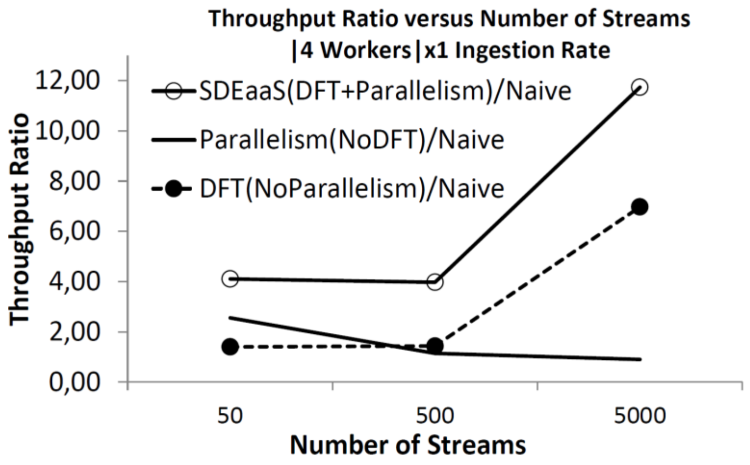


Figure 28: Thoughput Ration of our Method against other Candidates

This is important because, according to Figure 28, when we switch to monitoring 500 streams, i.e., 250K/2 similarity checks are performed by Naive, the fact that the Parallelism(NoDFT) approach lacks the ability of the DFT to bucketize time series and prune unnecessary similarity checks, makes its throughput approaching the Naive approach. This is due to the Aggregation operator starting to become a computational bottleneck for Parallelism(NoDFT) in the workflow. On the contrary, the DFT(NoParallelism) line remains steady when switching from 50 to 500 streams. The DFT(NoParallelism) approach starts to perform better than Parallelism(NoDFT) on 500 monitored streams showing that the importance of comparison pruning and, thus, of vertical scalability is higher than the importance of parallelism, as more streams are monitored. The line corresponding to our SDEaaS(DFT+Parallelism) approach exhibits steady behavior upon switching from 50 to 500, improving the Naive approach by 4 times, the DFT(NoParallelism) approach by 3 and the Parallelism(NoDFT) approach by 3.5 times.

The most important findings come upon switching to monitoring 5000 stocks (25M/2 similarity checks using Naive or Parallelism(NoDFT)). Figure 28 says that because of the lack of the vertical scalability provided by the DFT, the Parallelism(NoDFT) approach becomes equivalent to the Naive one. The DFT(NoParallelism) approach improves the throughput of the Naive and of Parallelism (NoDFT) by 7 times. Our SDEaaS(DFT+Parallelism) exhibits 11.5 times better performance compared to Naive, Parallelism(NoDFT) and almost doubles the performance of DFT(NoParallelism). This validates the potential of SDEaaS(DFT+Parallelism) to support interactive analytics upon judging similarities of millions of pairs of stocks. In addition, studying the difference between DFT(NoParallelism) and SDEaaS(DFT+Parallelism) we can quantify which part of the improvement over Naive, Parallelism(NoDFT) is caused due to comparison pruning based on time series bucketization and which part is yielded by parallelism. That is, the use of DFT for bucketization and dimensionality reduction increases throughput by 7 times (equivalent to the performance of DFT(NoParallelism)), while the additional improvement entailed by SDEaaS(DFT+Parallelism) is roughly equivalent to the number of workers (4 workers in Figure 28). This indicates the success of SDEaaS in integrating the virtues of data synopsis and parallel processing.

6 Conclusion & Future Work

In this work we introduced a Synopses Data Engine (SDE) for enabling interactive analytics over voluminous, high-speed data streams. Our SDE is implemented following a SDE-as-a-Service (SDEaaS) paradigm and is materialized via a novel architecture. It is easily extensible, customizable with new synopses and capable of providing various types of scalability. Moreover, we exhibited ways in which SDEaaS can serve workflows for different purposes and we commented on implementation insights and lessons learned throughout this endeavour

Our future work in the development of the SDE is concentrated towards the following directions:

• Incorporation of more synopsis and reduce techniques: the SDE library will be enhanced with a number of additional data summarization techniques that match the needs of a wide variety of applications. Carefully inspecting Table 2, one can easily deduce that at the current stage of the SDE library we have chosen, at least one technique for (distinct) count, frequency, set membership, cardinality, correlation, and sample estimation. These constitute broad categories of synopsis often used in various application scenarios. This list is going to be enhanced with other popular, approximate operators involving approximate top-K query answering and histogram computation. Moreover, more than one algorithm will be incorporated per category based on criteria and the variety of trade-offs they introduce compared to the already incorporated ones. Such criteria are related to mergeability, memory utilization, computational complexity of updating and querying the synopsis.

• Joins & Extended Windowing Support: the synopses we have currently incorporated, in large part operate over the whole stream history or disjoint windows of streams. In the future we plan to incorporate synopses that can equivalently support both tumbling (disjoint) and sliding windows. Moreover, we have designed a way that the current architecture could be used so as to implement the functionality of popular approximate join algorithms.

- Benchmarking: One of the main advantages of the SDE, besides providing techniques for reduced memory utilization, is that it can work on small, carefully-crafted portions (such as samples) of the data and provide rapid responses within the scope of complex workflows of Big Data analytics.

7 REFERENCES

[1] [n.d.]. Apache Beam v. 2.19.0. https://beam.apache.org/

[2] [n.d.]. Apache Flink v. 1.9. https://flink.apache.org/

[3] [n.d.]. Apache Kafka v. 2.3. https://kafka.apache.org/

[4] [n.d.]. Apache Spark v. 2.4.4. https://spark.apache.org/

[5] [n.d.]. Apache Storm v. 2.1. https://storm.apache.org/

[6] [n.d.]. Forbes. https://www.forbes.com/sites/tomgroenfeldt/2013/02/14/at-nyse-the-data-deluge-overwhelms-traditional-databases/#362df2415aab

[7] [n.d.]. Proteus Project. https://github.com/proteus-h2020

[8] [n.d.]. Stream-lib. https://github.com/addthis/stream-lib

[9] [n.d.]. Yahoo DataSketch. https://datasketches.github.io

[10] M. R. Ackermann, M. Märtens, C. Raupach, K. Swierkot, C. Lammersen, and C. Sohler. 2012. StreamKM++: A clustering algorithm for data streams. ACM Journal of Experimental Algorithmics 17, 1 (2012).

[11] P. K. Agarwal, G. Cormode, Z. Huang, J. Phillips, Z.Wei, and K. Yi. 2012. Mergeable Summaries. In PODS.

[12] N. Alon, Y.Matias, and M. Szegedy. 1996. The Space Complexity of Approximating the Frequency Moments. In STOC.

[13] B. Babcock, M. Datar, and R. Motwani. 2002. Sampling from a moving window over streaming data. In SODA.

[14] B. H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. Commun. ACM 13, 7 (1970), 422–426.

[15] M. Charikar. 2002. Similarity estimation techniques from rounding algorithms. In STOC.

[16] S. Chintapalli, D. Dagit, B. Evans, and et al. 2016. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In IPDPS Workshops.

[17] G. Cormode, M. Garofalakis, P. Haas, and C. Jermaine. 2012. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. Foundations and Trends in Databases 4, 1-3 (2012), 1–294.

[18] G. Cormode and M. N. Garofalakis. 2008. Approximate continuous querying over distributed streams. ACM Trans. Database Syst. 33, 2 (2008), 9:1–9:39.

[19] G. Cormode and S. Muthukrishnan. 2005. An improved data stream summary:the count-min sketch and its applications. J. Algorithms 55, 1 (2005), 58–75.

[20] G. Cormode and K. Yi. 2020 (to be published). Small Summaries for Big Data. Cambridge University Press.

[21] P. Flajolet, E. Fusy, O. Gandouet, and et al. 2007. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In AOFA.

[22] P. Flajolet and G. N. Martin. 1985. Probabilistic Counting Algorithms for Data Base Applications. J. Comput. Syst. Sci. 31, 2 (1985), 182–209.

[23] M. Garofalakis, J. Gehrke, and R. Rastogi. 2016. Data Stream Management: A Brave New World. In Data Stream Management - Processing High-Speed Data Streams.

[24] N. Giatrakos, E. Alevizos, A. Artikis, A. Deligiannakis, and M. N. Garofalakis. 2020. Complex event recognition in the Big Data era: a survey. VLDB J. 29, 1 (2020), 313–352.

[25] N. Giatrakos, N. Katzouris, and A. Deligiannakis et al. 2019. Interactive Extreme: Scale Analytics Towards Battling Cancer. IEEE Technol. Soc. Mag. 38, 2 (2019), 54–61.

[26] N. Giatrakos, Y. Kotidis, A. Deligiannakis, V. Vassalos, and Y. Theodoridis. 2013. In-network approximate computation of outliers with quality guarantees. Inf. Syst. 38, 8 (2013), 1285–1308.

[27] M. Greenwald and S. Khanna. 2001. Space-Efficient Online Computation of Quantile Summaries. In SIGMOD.

[28] Z. Kaoudi and J.A.Q.Ruiz. 2018. Cross-Platform Data Processing: Use Cases and Challenges. In ICDE.

[29] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In ICDE.

[30] G. S. Manku and R. Motwani. 2002. Approximate Frequency Counts over Data Streams. In VLDB.

[31] A. Milios, K. Bereta, K. Chatzikokolakis, D. Zissis, and S. Matwin. 2019. Automatic Fusion of Satellite Imagery and AIS Data for Vessel Detection. In Fusion.

[32] B. Mozafari. 2019. SnappyData. In Encyclopedia of Big Data Technologies.

[33] E. Zeitler and T. Risch. 2011. Massive Scale-out of Expensive Continuous Queries. PVLDB 4, 11 (2011).

[34] Y. Zhu and D. E. Shasha. 2002. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In VLDB.

[35] Rakesh Agrawal, Christos Faloutsos, Arun N. Swami: Efficient Similarity Search In Sequence Databases. FODO 1993: 69-84

[36] Nikolaos Pavlakis, Scaling out streaming time series analytics on Storm, MSc Thesis, Technical University of Crete, 2017, http://purl.tuc.gr/dl/dias/802769EF-66BD-411D-9E6E-BA00A61F2B6F

[37] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In G. D. Battista and U. Zwick, editors, European Symposium on Algorithms (ESA), volume 2832, pages 605--617, 2003.

[38] G.S. Manku, R. Motwani, Approximate frequency counts over data streams, in: VLDB, 2002, pp. 346–357.

[39] Graham Cormode and Minos Garofalakis. "Streaming in a Connected World: Querying and Tracking Distributed Data Streams" (tutorial abstract), Proceedings of VLDB'2006, 2006.

[40] Minos N. Garofalakis, Johannes Gehrke, and Rajeev Rastogi. "Querying and Mining Data Streams: You Only Get One Look" (tutorial abstract), Proceedings of ACM SIGMOD, 2002.

[41] Bonaventura Del Monte, Jeyhun Karimov, Alireza Rezaei Mahdiraji, Tilmann Rabl, Volker Markl: PROTEUS: Scalable Online Machine Learning for Predictive Analytics and Real-Time Interactive Visualization. EDBT/ICDT Workshops 2017.