And Synopses for All: a Synopses Data Engine for Extreme Scale Analytics-as-a-Service

Antonios Kontaxakis[a], Nikos Giatrakos[b,c,*], Dimitris Sacharidis[a], Antonios Deligiannakis[b,c]

[a]*Universite Libre de Bruxelles, CP 165/15, Avenue F.D. Roosevelt 50, 1050 Brussels, Belgium*
[b]*School of Electrical and Computer Engineering, Technical University of Crete, University Campus., 73100 Chania, Greece*
[c]*ATHENA Research and Innovation Centre, Artemidos 6 & Epidavrou, 15125 Athens, Greece*

Abstract

In this work, we detail 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. Our SDE is built on top of Apache Flink and implements a novel synopsis-as-a-service paradigm. In that, it achieves (i) concurrently maintaining thousands of synopses of various types for thousands of streams, on demand, (ii) reusing synopses that are common across various concurrent workflows, (iii) providing data summarization facilities even for cross-(Big Data) platform workflows, (iv) pluggability of new synopses on-the-fly, (v) increased potential for workflow execution optimization. The proposed SDE-as-a-service provides interactive analytics at scale by enabling 3 types of scalability: (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 across geo-distributed clusters and clouds by controlling the communication required to answer global queries.

1. Introduction

Real-time, 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 terrabytes of data per day, including trades of thousands of stocks streams [1] from a variety of markets. Stakeholders, such as authorities and investors, need to analyze these data in an interactive, online fashion for timely market surveillance or investment risk/opportunity identification purposes. 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 [2] 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, while in the maritime domain autonomous vehicles are added as on-site sensing information sources[1].

To enable real-time, online analytics at extreme-scale, three types of scalability are required:

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

*Corresponding author. Tel.: (+30) 28210 37265; Fax: (+30) 28210 37542
Email addresses: antonios.kontaxakis@ulb.be (Antonios Kontaxakis), ngiatrakos@softnet.tuc.gr, ngiatrakos@athenarc.gr (Nikos Giatrakos), dimitris.sacharidis@ulb.be (Dimitris Sacharidis), adeli@softnet.tuc.gr, adeli@athenarc.gr (Antonios Deligiannakis)

[1] https://www.liquid-robotics.com/

- 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 $\Theta(N^2)$ 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. A number of benchmarks [3, 4] conclude that, in such settings, the maximum achieved throughput (number of streaming tuples that are processed per time unit) is network bound. Indeed, moving the raw data streams around sites (data centers collecting local market data or ground stations in the financial and maritime scenario, respectively) in order to accomplish a geo-distributed analytics task, depletes the available bandwidth introducing network latencies that prevent interactivity.

To handle the volume and velocity of Big streaming Data, Big Data platforms such as Apache Flink or Apache Spark have designed dedicated APIs to facilitate scaling-out, i.e., parallelizing, the computation of streaming analytics tasks to a number of Virtual Machines (VM) available in corporate computer clusters or the cloud. However useful these facilities may be, they only focus on a narrow part of the challenges analytics workflows need to encounter for stream processing at scale. This is because the mere use of parallelism in Big Data platforms only partially supports horizontal scalability, while vertical and federated scalability are completely neglected.

On the other hand, there is a wide consensus in stream processing [5, 6, 7, 8, 9, 10, 11] 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 sufficient 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 [9] build carefully-crafted synopses of Big streaming Data which preserve data properties important for providing approximate answers, with tunable 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 [9].

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 [12] 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 [5], Locality Sensitive Hashing (LSH) sketches [13, 14] or the number of set bits (a.k.a. Hamming Weight) in LSH bitmaps [15, 16] have been used for correlation/distance-aware hashing of streams to respective processing units. Based on the synopses, using DFT coefficients, LSH-based sketches or Hamming Weights as the hash key respectively, highly uncorrelated/dissimilar 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 [17] of many synopses techniques. As an example, answering cardinality estimation queries over a number of sites, each maintaining its own FM sketch [9] is as simple as communicating only small bitmaps (typically 64-128 bits) to the query source and performing a bitwise `OR` operation.

Surprisingly, Big Data platforms strongly support stream processing, but provide no API dedicated to synopses construction and maintenance. In other words, synopses-centered APIs are significantly missing. To fill this gap, in this work, we detail the design and structure of a Synopses Data Engine (SDE) [18] built on top of Apache Flink. 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. Our SDE design (i) implements a novel Synopses-as-a-Service (termed SDEaaS) paradigm, (ii) is already incorporated in a commercial analytics platform [19], namely RapidMiner Studio, (iii) is available open-source [20] and (iv) has been put into production in various real-world applications [21, 22].

More precisely, our contributions are:

1. We present the novel architecture of a Synopses Data Engine (SDE) capable of providing interactivity in extreme-scale analytics by enabling various types of scalability.
2. Our engine introduces a novel SDE-as-a-Service (SDEaaS) paradigm according to which, the SDE is a constantly running job in one or more clusters/clouds accepting on-the-fly requests for (i) plugging-in the code of new synopses definitions (ii) starting maintaining new synopses (iii) querying maintained synopses in an ad-hoc and/or continuous fashion. SDEaaS also accounts for synopses-sharing, avoiding to duplicate streams and synopses that are common in multiple, broader workflows.
3. We describe 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. We discuss insights we gained while materializing a SDEaaS paradigm and outline lessons learned useful for future, similar endeavors.
5. We showcase how the proposed SDE can be used in workflows to serve a variety of purposes towards achieving interactive data analytics.
6. We present 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.

2. Related Work

From a research viewpoint, there is a large number of related works on data synopsis construction and maintenance techniques. Please refer to [9, 10, 11] for comprehensive reviews on relevant algorithms. All these approaches constitute our algorithmic arsenal and Table 1 summarizes stream synopsis techniques that are already incorporated in our SDE while, some of them, are further discussed in the case study of Section 7.

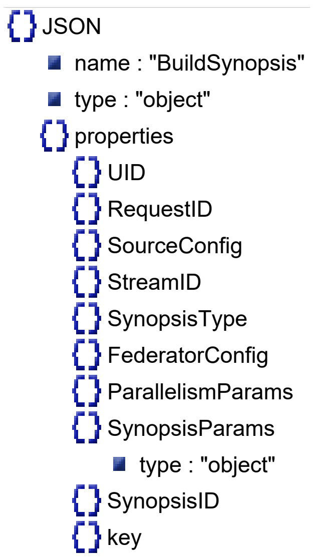
Few libraries and synopsis APIs have been developed in prior efforts. Apache DataSketches [23] and Stream-lib [24] are software libraries of stochastic streaming algorithms and summarization techniques, correspondingly. These libraries are detached from parallelization/ horizontal scalability aspects. SnappyData's [25] stream processing is based on Spark and incorporates a limited set of synopses serving simple `SUM`, `COUNT` and `AVG` queries. Similarly, StreamApprox [26] offers only sampling as a pipeline operator. Thus, these are deprived of vertical scalability features and federated scalability provisions. The recent, prominent work of Condor [27] elegantly optimizes the parallel computation of stream summaries over Flink, still neglecting aspects of horizontal scalability. More precisely, Condor needs to start a new job and reserve at least one entire task slot (thread) in Flink for each request of maintaining a new synopsis. What our SDEaaS does instead, is to create new tasks at the runtime of a unified synopses maintenance job. In that, our experimental evaluation shows that we can maintain thousands of synopses for thousands of streams in setups where all the aforementioned approaches can run only few tens of synopses. Moreover, Condor does not account for vertical and federated scalability.

Sampling		
Synopsis	*Output Estimation*	*Configuration Parameters*
Distributed Sampling [28]	Sample	Sample Size, W/Wo Replacement
Windowed Distributed Sampling [28]	Sample	Window Size, Sample Size
STSampler [21]	Trajectory Sample	Angle, Direction, Distance, Time Interval Thresholds
Chain Sampler [29]	Sample	Sample Size
Counting		
Synopsis	*Output Estimation*	*Configuration Parameters*
HyperLogLog Sketch [8]	Distinct Count	Relative Standard Error
FM Sketch [30]	Distinct Count	Bitmap Size, ϵ, δ
CountMin [7]	Count/Frequency Estimation	ϵ,δ
Lossy Counting [6]	Count, Frequent Items	ϵ
Sticky Sampling [6]	Count, Frequent Items	Support, ϵ, δ
BloomFliter [31]	Set Membership	#elements, False Positive Rate
Correlation/Norm Computation		
Synopsis	*Output Estimation*	*Configuration Parameters*
AMS Sketch [32]	L_2-Norm, Inner Product	ϵ, δ
CoreSetTree [33]	CoreSets	Bucket Size, Dimensionality
Discrete Fourier Transform (DFT) [5]	Correlation Score, DFT Coefficients Hash Key/BucketID	Similarity Threshold, Number of Coefficients
Random Hyperplane Projection (RHP) [34, 15]	Correlation Score Hash Key/BucketID	Similarity Threshold, Bitmap Size, Number of Buckets
Radius Sketch Family [13, 14]	List of Streams	Group & Sketch Sizes, Threshold, Window Size, Number of Groups
Quantiles		
Synopsis	*Output Estimation*	*Configuration Parameters*
GKQuantiles [35]	Quantiles	ϵ
AMQuantiles [36]	Quantiles	Window Size, ϵ

Table 1: Supported synopses. ϵ is the approximation error bound. δ is the probability of failing to achieve ϵ accuracy. For synopses that can be maintained over a window, respective parameters for window definition are added.

3. SDE API – Supported Operations

In this section, we outline the functionality that our SDE API provides to upstream (i.e., contributing input to) and downstream (receiving input from) operators and application interfaces of a given Big Data processing workflow engaging synopses. All requests are submitted to the SDE at runtime, given the SDEaaS nature of our design, via lightweight, properly formatted JSON snippets [37] to ensure cross-(Big Data) platform compatibility. The JSON snippet of each request listed below includes a unique identifier for the queried stream or data source incorporating multiple streams (see Section 4) and a unique id is created for every loaded/created/queried synopsis. In case of a create or load synopsis request (see below and Table 1), the parameters of the synopsis as well as a pair of parameters involving the employed parallelization degree and scheme (see Section 4) are also included in the JSON snippet. In federated architectures where multiple, geo-dispersed clusters run local SDEaaS instances and estimations provided



```
{ } JSON
    name : "BuildSynopsis"
    type : "object"
  { } properties
      { } UID
      { } RequestID
      { } SourceConfig
      { } StreamID
      { } SynopsisType
      { } FederatorConfig
      { } ParallelismParams
      { } SynopsisParams
            type : "object"
      { } SynopsisID
      { } key
```

Figure 1: JSON snippet for `BuildSynopsis` request.

by synopses need to be collected at a single cluster afterwards, the JSON snippet also includes the (IP address, port) of that cluster.

Our SDE communicates with upstream and downstream operators via Kafka [38]. The SDE API accepts the following requests (via the `RequestTopic` in Figure 2):

Build/Stop Synopsis. A synopsis can be created or ceased on-the-fly, as the SDE is up and running. In that, the execution of the rest of the running workflows that utilize other synopsis operators, is not hindered. A synopsis may be (i) a single-stream synopsis, i.e., a synopsis (e.g. sample) maintained on the trades of a single stock, or (ii) a data source synopsis, i.e., a synopsis maintained on all trades irrespectively of the stock. Moreover, `Build/Stop Synopsis` allows submitting a single request for maintaining synopses of the same kind, for each out of multiple streams coming from a certain source. For instance, maintaining a sample per stock for thousands of stocks coming from the same data source requires the submission of a single request. A condensed view of a JSON snippet for building a new synopsis is illustrated in Figure 1.

Load Synopsis. The SDE Library (Table 1, Section 5.1) incorporates a number of synopsis operators, commonly used in practical scenarios. `Load Synopsis` supports pluggability of the code of additional (not included in the SDE Library) synopses, their dynamic loading and maintenance at runtime. The structure of the SDE Library, utilizing inheritance and polymorphism, is key for this task. This is an important feature since it enables customizing the SDE to application specific synopses without stopping the service.

Ad-hoc Query. The SDE accepts one-shot, ad-hoc queries on a certain synopsis and provides respective estimations (approximate answers) to downstream operators or application interfaces, based on its current status.

Continuous Querying. Continuous queries can be defined together with a `Build/Stop Synopsis` request. In this case, an estimation of the approximated quantities, such as counts, frequency moments or correlations are provided every time the estimation of the synopsis is updated, for instance, due to reception of a new tuple.

The response to ad-hoc or continuous queries is also provided in lightweight JSON snippets including: (i) a `<key, value>` pair for uniquely identifying the provided response (e.g., from past and future ones) and for the value of the estimated quantity, respectively, (ii) the identifier of the request that generated the response, (iii) the identifier of the utilized synopses along with its configuration parameters (Table 1).

`SDE Status Report`. The API allows querying the SDE about its status, returning information about the currently maintained synopses and their parameters. This facility is useful during the definition of new workflows, since it allows each application to discover whether it can utilize already maintained data synopses and reuse each synopsis in multiple workflows.

4. SDE Architecture

In this section we detail the SDE architectural components and present their utility in serving the operations specified in Section 3.

4.1. SDE Fundamentals

Our architecture is built on top of Apache Flink [39] and Kafka [38, 40]. Kafka is used as a fast, scalable, durable, and fault-tolerant publish-subscribe messaging system enabling connectivity between the SDE and upstream, downstream operators in the workflows served by the SDE. Kafka together with the JSON format of accepted request snippets allows us to materialize the SDEaaS paradigm even when upstream or downstream operators of broader workflows run on Big Data platforms other than Flink. Furthermore, Kafka is used as a messaging service in case of querying synopses maintained at a number of geo-dispersed clusters. A Kafka cluster is composed of a number of brokers, run in parallel, that handle separate partitions of topics. Topics constitute categories of data where producers and consumers can write and read, respectively. In the case of the SDE, producers constitute upstream operators, while downstream operators act as consumers. Furthermore, in geo-dispersed, multi-cluster settings, the SDE instances running at different clusters may be the producers or consumers of a particular Kafka topic as will be explained later on in this section.

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 [39] 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. Figure 2 provides a condensed view of the SDEaaS architecture, which engages `Map`, `FlatMap`, `CoFlatMap`, `Union` and `Filter` Flink operators (stream transformations). In a nutshell, a `Map` operator takes one tuple and produces another tuple in the output depending on the functions it executes on the data, a `FlatMap` operator takes one tuple and produces zero, one, or more tuples, while a `CoFlatMap` operator hosts two `FlatMap` that share access to common variables (therefore the linking icon in the figure) among streams that have previously been connected (using a `Connect` in Flink [39]). `Filter` evaluates a boolean function for each tuple and retains those tuples for which the function returns true. Finally, a `Union` operator receives two or more streams and creates a new one containing all their elements. Section 4.2 explains the reason for the above design and the flow of information in different uses of the SDE.

4.2. SDE Architectural Components

Employed Parallelization Schemes. The parallelization scheme that is employed in the design of the SDE is partition-based parallelization [41]. That is, every data tuple that streams in the SDE architecture and is destined to be included in a maintained synopsis, does so based on the partition key assigned to it[2]. When a synopsis is maintained for a particular stream (for instance, per stock) the key that is assigned to the respective update (newly arrived data tuple) is the identifier of that particular stream for which the synopsis is maintained. In this case, within the distributed computation framework of Flink, that stream is processed by a task of the same worker and parallelization is achieved by distributing the number of streams for which a synopsis is built, to the available workers in the cluster(s) hosting the SDEaaS. On the other hand, when a synopsis involves a data source (for instance, a data source for all monitored stock streams of a particular, regional stock market), the desired degree of parallelism is included as a parameter in the respective `Build Synopsis` request for the synopsis. 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 SDEaaS (details on that follow shortly) each of which points (is hashed) to a particular worker. Finally, in case of processing streaming windows (either time- or count-based) [41] an incoming tuple may (i) initiate a new window, (ii) be assigned to one or more existing windows or (iii) terminate a window. Here, the partition is the window itself and the tuple is given the key(s) of the window(s) it affects.

[2]`KeyBy` transformations (not shown in Figure 2 for readability purposes) are used to partition streams and data sources, i.e., all elements with the same key are assigned to the same partition.

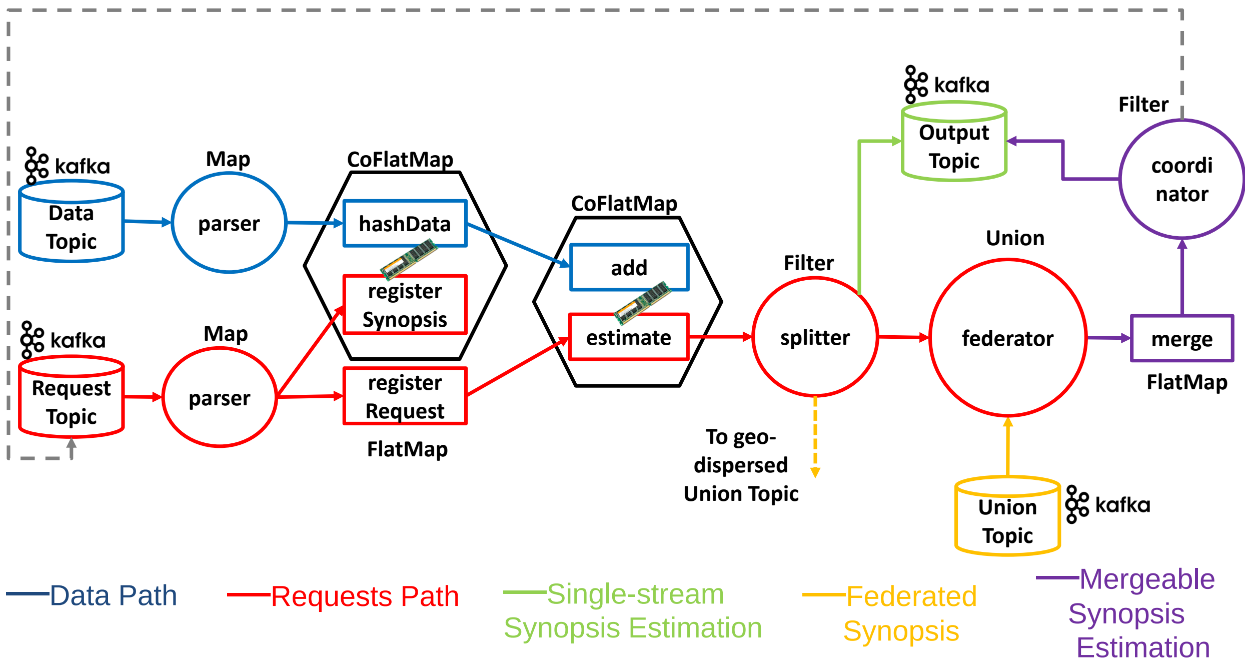


Figure 2: SDEaaS Architecture – Condensed View

In Figure 2 we include outside of the depicted shapes, the name of the Flink stream transformations [39] that are programmatically used. Inside each shape, we provide a declarative name of that transformation to denote the functionality it executes. This is to enhance our description with the semantics of each stream transformation. Each operator actually executes in multiple parallel instances, Figure 2 provides only a condensed view (i.e., parallel instances for each operator are not explicitly shown) to improve readability.

Data and Query Ingestion. Data and request (JSON snippet) streams arrive at a particular Kafka topic each. In the case of the `DataTopic` (blue-colored) of Figure 2, a parser (`Map`) component is used in order to extract the key and value field(s) on which a currently running synopsis is maintained. The respective parser (`Map`) of the `RequestTopic` (red-colored) topic of Figure 2 reads the JSON snippet of the request and processes it as follows: When an incoming request (Section 3) is a `Build Synopses` request, the parser component extracts information about the parameters of the synopsis (as detailed in Table 1) and its nature, i.e. whether it is on a single stream, on a data source, if it involves a multi-stream synopsis maintenance request or a synopsis that is also maintained in SDE instances in other geo-dispersed clusters. In case of an `Ad-hoc Query` request, the parser component extracts the queried synopsis identifier(s).

Requesting New Synopsis Maintenance. When a `Build Synopses` request is issued, it follows part of the red-colored path of the SDE architecture in Figure 2. That is, the corresponding parser sends the request to a `FlatMap` operator (termed `registerRequest` at the bottom of Figure 2) and to another `FlatMap` operator (`registerSynopsis`) which is part of a `CoFlatMap` one. `registerRequest` and `registerSynopsis` produce the same keys (as analyzed in the description of the supported parallelization schemes) for the maintained synopsis, but provide different functionality. The `registerRequest` operator uses these keys in order to later decide which worker(s) an ah-hoc query, which also follows the red-colored 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, which follows the blue-colored path in Figure 2, should be directed.

Updating the Synopsis. When a data tuple destined to update one or more maintained synopses is ingested via the `DataTopic` of Kafka it follows the blue-colored path of the SDE architecture in Figure 2. The tuple is directed to the `hashData` `FlatMap` of the corresponding `CoFlatMap` where the keys (stream identifier for single stream synopsis and/or worker identifier for data source synopses and windowing operations) 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 FM sketch [30] 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.
Ad-hoc Query Answering. An ad-hoc query arrives via the RequestTopic of Kafka and is directed to the register Request operator. The operator which has produced the keys using the same code 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 (middle of Figure 2). The estimate operator reads, via the shared state of CoFlatMap, 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 request on a FM sketch [30], the estimate operator reads the maintained bitmap, finds the lowest position of the unset bit and provides a distinct count estimation by using the index of that position and a $\phi = 0.77$ coefficient. Table 1 summarizes the estimated output quantities each of the currently supported synopses can provide.
Continuous Query Answering. In case continuous queries are to be executed on the maintained synopses, a new estimation needs to be provided every time the estimation of the synopsis is updated, either via an add operation or because a window on the data expires. In this particular occasion, estimate needs to be invoked directly when add is executed.

Both in Ad-hoc Query and Continuous Querying, the result of estimate, following the red path in Figure 2, is directed to a Filter operator, termed splitter. If necessary, the splitter forwards estimations to a Union operator, termed federator which reads from a Union Kafka topic (yellow path in Figure 2). The Union Kafka topic and the federator involve our provisions for maintaining federated synopses, i.e., synopses that are kept at a number of potentially geo-dispersed clusters where instances of the SDEaaS run. The splitter distinguishes between three cases.

- **Case 1:** Case 1 happens when estimate involves a single-stream synopsis maintained only locally at a cluster. Then, splitter directs the output to downstream operators of the executed workflow via the Kafka OutputTopic, by following the green-colored path in Figure 2.
- **Case 2:** Case 2 arises when a federated synopsis is queried but the request has identified another cluster's (IP address, port) responsible for extracting the overall estimation. Then, splitter acts as the producer (writes) to the geo-dispersed Union Kafka topic of another cluster (declared by the dotted, yellow arrow coming out of splitter in Figure 2).
- **Case 3.1:** For non-federated synopses defined on entire data sources (e.g., a sample over all stock data of a regional market), a number of workers of the current cluster participate in the employed parallelization scheme as explained at the beginning of Section 4.2. Thus, each such worker provides its local estimation output. Because something similar holds when individual clusters maintain federated synopses and the current cluster is set as responsible for synthesizing the overall estimation, in both cases the output of the splitter operator is directed via the federator (Union Flink operator) to a merge FlatMap following the purple-colored path. The merge operator merges the partial results of the various workers and/or clusters and produces the final estimation which is streamed to downstream operators, again via an Output Kafka topic. For instance, FM sketches [30] or Bloom Filters [31] (bitmaps) can be merged via simple logical disjunctions or conjunctions. At this point, in order to direct all partial estimates to the same worker of a cluster to perform the merge operation, a corresponding identifier for the issued request (for ad-hoc queries) or an identifier for the maintained synopsis (for continuous queries) is used as the key. However, the merge operator does not directly act as a producer for the Output Kafka topic, but through a coordinator Filter operator. Let us explain why.
- **Case 3.2:** Our SDEaaS supports synopses, Distributed Sampling [28] and Windowed Distributed Sampling [28] in Table 1, which require bidirectional communication to both perform the merge of partial results and send back to the workers some information, for instance, for changing sampling rounds as entailed by the techniques in [28]. In such cases, the coordinator in Figure 2 besides writing the current sample to the Output Kafka topic, it also writes the required info to the RequestTopic as illustrated by the dotted gray arrow in Figure 2. This is required because the processing graphs in Flink are supposed to be Directed Acyclic Graphs (DAGs) and, thus, no operator can propagate its output backwards.

5. SDE Library & SDEaaS GUI

5.1. *The SDE Library*

The internal structure of the synopses library is illustrated in Figure 3 which provides only a partial view of the currently supported synopses for readability purposes. Table 1 provides a full list of currently supported synopses, their utility in terms of output estimations and configuration parameters. The development of the SDE Library exploits subtype polymorphism in Java in order to ensure the desired level of pluggability for new synopses definitions.

As shown in Figure 3, there is a higher level class called `Synopsis` with attributes related to a unique identifier and a couple of strings. The first string holds the details of the request (JSON snippet) with respect to how the synopsis should be physically implemented, i.e., index of the key field in an incoming data tuple (for single stream synopsis), the respective index of the value field which the synopsis is built on, whether the synopsis is a federated one and which cluster should synthesize the overall estimation and so on. The second string holds information included in the JSON snippet regarding synopsis configuration parameters as those cited in Table 1. Furthermore, the `Synopsis` class includes methods for `add`, `estimate` and `merge` as those were described in Section 4. Finally, a set of setters and getters for synopsis, key and value identifiers are provided.

Every specific synopsis algorithm is implemented in a separate class, as shown in Figure 3, that extends `Synopsis` and overrides the `add`, `estimate` and `merge` methods with the algorithmic details of that particular technique as cited in Table 1.

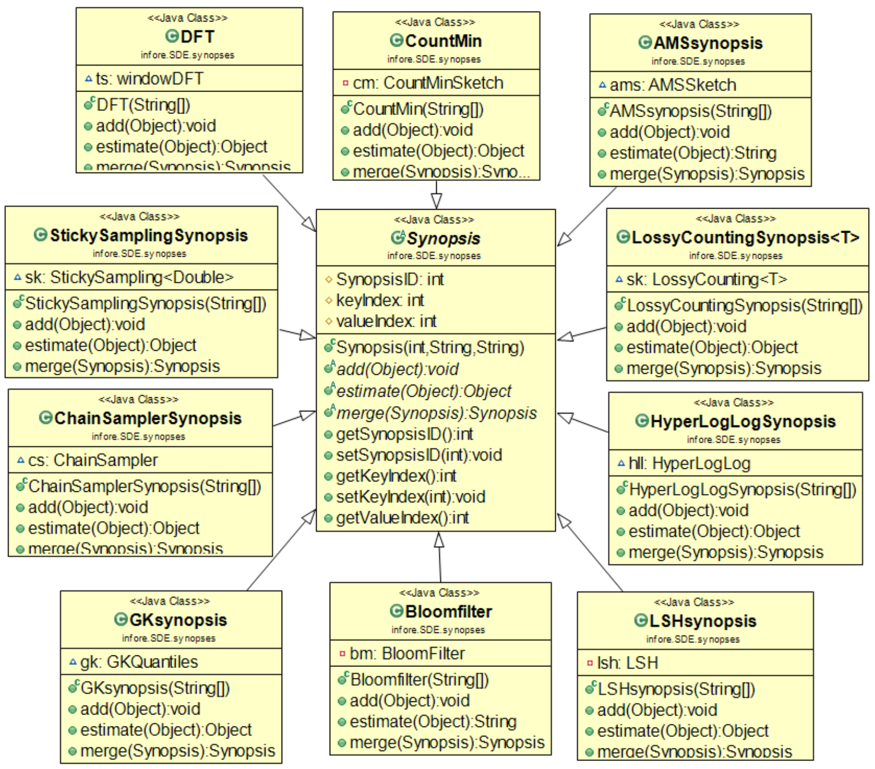
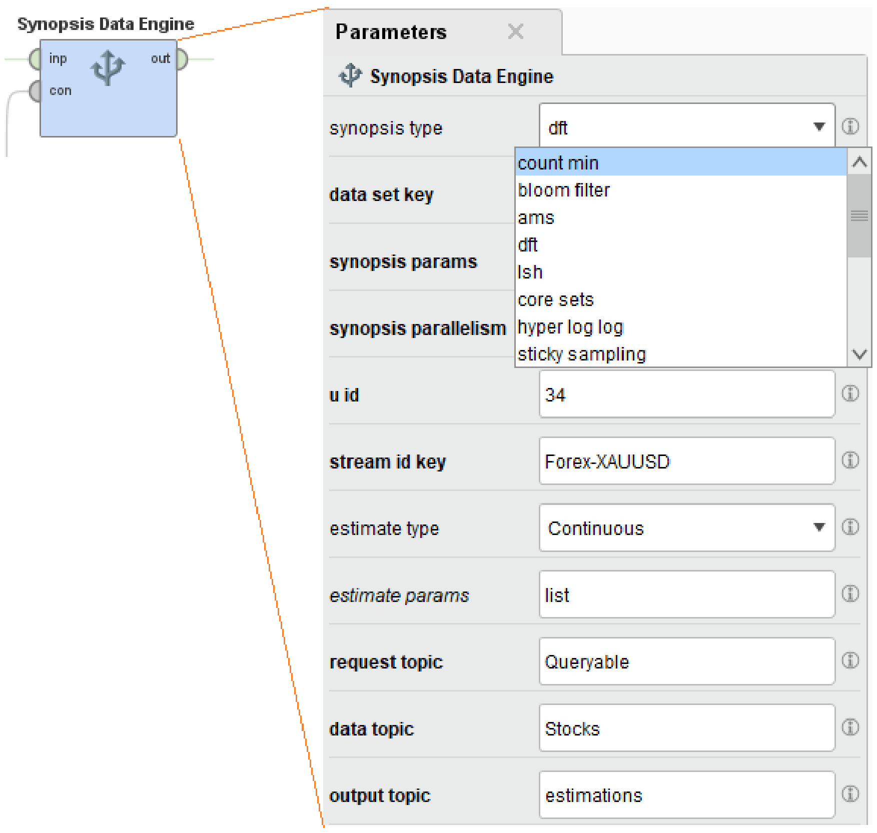


Figure 3: Structure of the Synopses Library (partial view).

5.2. *The SDE GUI*

Our SDEaaS has already been incorporated in the streaming extension [19] of a commercial data analytics platform, namely RapidMiner Studio[3]. The basic concept in RapidMiner Studio is to design and execute advanced

[3]`https://rapidminer.com/`



Synopsis Data Engine

synopsis type: dft
data set key
synopsis params
synopsis parallelism
u id: 34
stream id key: Forex-XAUUSD
estimate type: Continuous
estimate params: list
request topic: Queryable
data topic: Stocks
output topic: estimations

Figure 4: SDEaaS as an operator in RapidMiner Studio.

analytics workflows in a code-free way. In that spirit, each operator that can be incorporated in a Big Data processing workflow can be dragged and dropped on a design canvas. Operators are then connected by lines to define the data flow.

The SDEaaS, i.e., the whole architecture extensively presented in Section 4 and Figure 2, is represented by a simple box in RapidMiner studio, as illustrated in Figure 4. As shown on the left of Figure 4 the SDEaaS operator possesses two input ports labeled `inp` and `con`, respectively. The `inp` is where the upstream operator (contributing input to the SDE) is expected to get connected, while the `con` accepts a connection object to one of the Flink clusters which run SDEaaS instances. The `out` port provides output to downstream operators of a workflow.

The synopsis that will be executed by each such box is defined in a parameters tab appearing on the right of Figure 4. The synopsis available in the SDE Library (the lot of synopses mentioned in Table 1) appear in a drop-down list, Kafka topics (`DataTopic`, `RequestTopic`, `OutputTopic` etc) required by the architecture of Figure 2 are declared in corresponding textboxes and JSON parameters as illustrated in Figure 1 are graphically presented along with default (but editable) values.

We showcase the usage of the SDEaaS in a broader workflow in Figure 5. The depicted workflow aims at analyzing stock data and discover cross-correlations among pairs or groups of stocks. It uses 2 synopses (`SDE.DFT`, `SDE.AMS` in the figure) maintained by two separate SDEaaS instances running on different Flink clusters. Further details about the application scenario are provided in Section 7. The yellow-colored notes under each operator in Figure 5 mainly involve the correspondence between the operators as designed in RapidMiner Studio and those discussed in Section 7 and Figure 6.

Nonetheless, note that our SDEaaS is not executable exclusively via the RapidMiner Studio. Instead, it can be used independently within any Big Data processing workflow, since our open source repository [20] provides a parameterizable client [42] for automating the set up of the engine. In fact, in our case study (Section 7) and

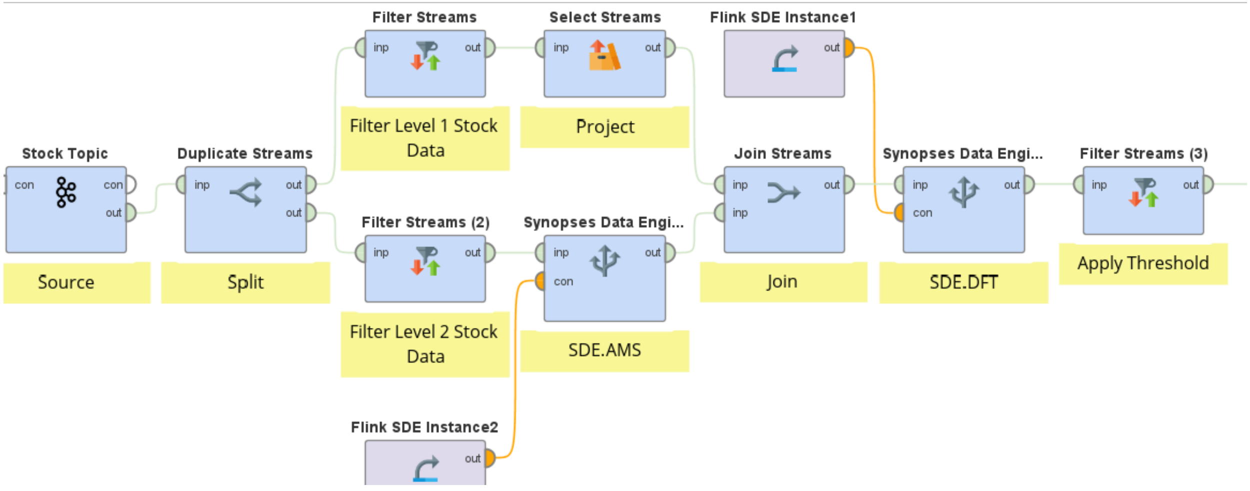


Figure 5: Workflow of Plan 3 (Section 7) in RapidMiner Studio, Synopsis-based Optimization for Enhanced Horizontal Scalability.

experimental evaluation (Section 8) we program the workflows outside RapidMiner Studio. Furthermore, recall again that SDEaaS can participate in workflows that are not programmed on Flink since it communicates with upstream and downstream operators only via Kafka.

6. Insights and Lessons Learned

Why Flink. In principle, our architectural design can be materialized over other platforms and APIs such as Spark or Kafka Streams. The key reason for choosing Flink as the platform for a proof-of-concept implementation of the proposed architecture is the `CoFlatMap` operator (stream transformation). As shown in the description of our architecture, the fact that `CoFlatMap` allows two `FlatMap` operators gain access to shared variables was used both for generating keys and assign data to partitions processed by certain workers (leftmost `CoFlatMap` in Figure 2) as well as for querying maintained synopses via the `estimate FlatMap` in the middle of the figure. Although one can manually program the `CoFlatMap` functionality in other Big Data platforms, the native support provided by Flink alleviates the development effort with respect to memory configuration, state management and fault tolerance.
The Red Path. Notice, that the blue-colored path in Figure 2 remains totally detached from the red-colored path. This depicts a design choice we follow for facilitating querying capabilities. That is, since the data updates on several maintained synopses may be ingested at an extremely high rate in Kafka at the beginning of the blue path, typically a lot higher than the rate at which requests are issued in the red path, in case the two paths were crossing, back-pressure on the blue-colored path would also affect the timely answers to requests. By having kept the two paths detached, requests can be answered in a timely manner based on the current status of the maintained synopses.
One SDE-as-a-Service For All. Our SDEaaS approach allows the concurrent maintenance of thousands of synopses for thousands of streams on demand (Section 8). It further allows different application workflows to share and reuse existing synopses instead of redefining them. The alternative is to submit a separate job with each (one or more) desired synopsis being part of a respective workflow that uses it. The latter simplistic approach possesses a number of drawbacks. First, the same synopses, even with the exact same parameters, cannot be reused/shared among currently running workflows. This means that data streams need to be duplicated and redundant data summaries are built as well. Second, one may end up submitting a different job for each new demand for a maintained synopsis. Apart from increasing the load of a cluster manager, this poses restrictions on the number of synopses that can be simultaneously maintained. Recall from Section 4.1 that each worker in a Flink cluster is assigned a number of task slots and each task slot can host tasks only of the same job. Therefore, lacking our SDEaaS approach means that the number of concurrently maintained synopses is at most equal to the available task slots. As a rule-of-thumb [39], a default number of task slots would be the number of available CPU cores. As we demonstrate in Section 8, SDEaaS can indeed maintain thousands of synopses for thousands of streams with only few tens of cores. In the same setup, all previous

approaches (Section 2) which lack the synopses-as-a-service paradigm can maintain only few tens of synopses. How do we achieve that and how the SDEaaS architecture avoids limitations of other approaches? In SDEaaS a request for a new synopsis on-the-fly, at runtime, assigns new tasks for the new synopsis in a continuously running job. On the contrary, lacking the SDEaaS rationale assigns at least one entire task slot to a new job. In SDEaaS, synopses maintenance involves tasks running instances of the operators in Figure 2, instead of devoting entire task slots to each. Each synopsis by design consumes limited memory and entails simple update (`add` in Figure 2) operations. Thus, in SDEaaS, we have multiple, lightweight tasks virtually competing for task slot resources and better exploit the potential for hyper-threading and pseudo-parallelism for the maintained synopses. For the above reasons, SDEaaS is a much more preferable design choice (also see Section 8).

Kafka Topics. In Figure 2 we use five specific Kafka topics which the SDE consumes (`DataTopic`, `RequestTopic`, `UnionTopic`) or produces (`OutputTopic`, `UnionTopic`). Our SDE is provided as a service and constantly runs as a single Flink job (per cluster, in federated settings). Synopses are created and respective sources of data are added on demand, but our experience in developing the proposed SDE says that there is no reliable way of adding/removing new Kafka topics to a Flink job dynamically, at runtime. Therefore all data tuples, requests and outputs need to be written/read in the respective data topics, each of which may include a number of partitions, i.e., per stream or data source. This by no means introduces redundancy in the data/requests processed by the SDE, because every data tuple that arrives in the `DataTopic` has no reason of existing there unless it updates one or more maintained synopses. Similarly every request that arrives in the `RequestTopic` creates/queries specific synopses. The same holds for the `OutputTopic` and `UnionTopic`. No output is provided unless a continuous query has been defined for a created synopses or an ad-hoc request arrives. In both cases, the output is meant to be consumed by respective application workflows. Furthermore, internal to the SDE, nothing is consumed or produced in the `UnionTopic` unless one or more federated synopses are maintained.

Windows & Out-of-order Arrival Handling. In Flink, Spark and other Big Data platforms, should a window operator need to be applied on a stream, one would use a programming syntax similar to (`{streamName||operatorName}.chosenWindowOperator`). If one does that in a SDEaaS architecture, the window would be applied to the entire operator, i.e., `CoFlatMap`, `FlatMap` and so on in Figure 2. But, in the general case, each maintained synopsis incorporates the definition of its own window which may differ across different currently maintained synopses, instead of the same window operator applied to all synopses. Therefore, a SDEaaS design does not allow for using the native windowing support provided by the Big Data platform because the various windows are not known in advance. One should develop custom code and exploit low-level stream processing concepts provided by the corresponding platform (such as the `ProcessFunction` in Flink [39]) to implement the desired window functionality. The same holds for handling out-of-order tuple arrivals and the functionality provided by `.allowedLateness()` in Flink or similar operators in other platforms.

Dynamic Class Loading. YARN-like cluster managers, upon being run as sessions, start the TaskManager and JobManager processes with the Flink framework classes in the Java classpath. Then job classes are loaded dynamically when the jobs are submitted. But what we require in a `Load Synopsis` request provided by our API is different. Due to the SDEaaS nature of the SDE, to materialize `Load Synopsis` we need to achieve loading classes dynamically *after* the SDE job has been submitted, as the service is up and running. A cluster manager will not permit loading classes at runtime due to security issues, i.e. class loaders are to be immutable. In order to bypass such issues for classes involving synopses that are external to our SDE Library, one needs to store the corresponding `jar` file in HDFS and create an own, child class loader. That is, the child class loader must have a constructor accepting a class loader, which must be set as its parent. The constructor will be called on JVM startup and the real system class loader will be passed. We leave testing `Load Synopsis` using alternative ways (e.g., via REST API), for future work.

7. SDEaaS Case Studies

In this section we design a specific scenario, we build a workflow that resembles, but extends, the Yahoo! Benchmark [37] and then, we discuss how our SDE and its SDEaaS characteristics can be utilized so as to serve a variety of purposes. Consider our running example from the financial domain. The workflow of Figure 6 illustrates a scenario that utilizes Level 1 and Level 2 stock data aiming at discovering cross-correlations among stocks. More precisely, Level 1 data involve stock trades of the form $< Date, Time, Price, Volume >$ for each 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

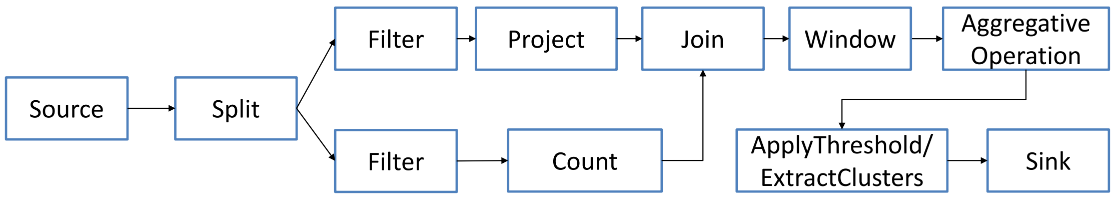


Figure 6: SDEaaS in Practice – Workflow under Study.

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/bid happens. The higher the number of such pairs for a stock, the higher the popularity of the stock. Note that, in Figure 6, we use generic operator namings. The workflow may be specified in any Big Data platform, other than Flink, and still use (in ways that we describe here) the benefits of SDEaaS acting as producer (issuing requests) and consumer to the Kafka topics of Figure 2, abiding by the respective JSON schemata.

In Figure 6 both Level 1 and Level 2 data arrive at a `Source`. The `Split` operator separates Level 1 from Level 2 data. It directs Level 2 data to the bottom branch of the workflow. There, the bids are `Filtered` (i.e., for monitoring only a subset of stocks or keep only bids above a price/volume threshold). Then, the bids are `Counted` and only this counter is kept per stock. When a trade for a stock is realized, the corresponding Level 1 tuple is directed by `Split` to the upper part of the workflow. A `Project` operator keeps only the timestamp and price of the trade for each stock. The `Join` operator afterwards joins the stock trade, Level 1 tuple with the count of bids the stock received until the trade. The corresponding result is inserted in a time `Window` of recent such counts, forming a time series. The pairwise similarities of the time series or coresets [33] of stocks are computed via an `AggregativeOperation`. The results either in the form of pairs of stocks surpassing a similarity threshold (`ApplyThreshold` operator in Figure 6) or clusters of stocks (`ExtractClusters` operator in Figure 6) are directed to a `Sink` to support relevant decision making procedures.

SDEaaS as a Cost Estimator for Enhanced Horizontal Scalability. SDEaaS can act as a cost estimator that constantly collects statistics for streams (in this scenario, stocks) that are of interest and these statistics can be used for optimizing the execution of any currently running or new workflow [22]. In our examined scenario, having designed the workflow in Figure 6 we wish to determine an appropriate number of workers that will be assigned for its execution, prescribing the parallelization degree, as well as balance the processing load among the dedicated workers. For that purpose a HyperLogLog [8] and a CountMin [7] sketch (see Table 1) can be used, i.e., our SDE constantly runs as a service and keeps HLL and CountMin sketches.

HyperLogLog (HLL) sketches [8] enable the extraction of approximate distinct counts using limited memory and a simple error approximation formula. Therefore, they are useful for estimating the cardinality of the set of stocks that are being monitored per time unit. 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 an equal probability of occurring each. The first m bits of the bitmap are used for bucketizing an incoming element and we have an array M of 2^m buckets (also called registers). The rest $64 - m$ 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, one needs to compute the harmonic mean of the values of the buckets. The relative error of HLL in the estimation of the distinct count is $1/\sqrt{2^m}$. HLL are trivial to merge based on equivalent number of buckets maintained independently at each site/cluster. One should simply derive the maximum among the corresponding buckets of sites.

A CountMin Sketch [7] 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, $w = e/\epsilon$ (e is the Eurler's number) and $d = log(1/\delta)$. d random, pairwise independent hash functions are chosen for hashing each tuple (concerning a particular stock) to a column in the sketch. When a tuple 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, when N is the sum of all frequencies so

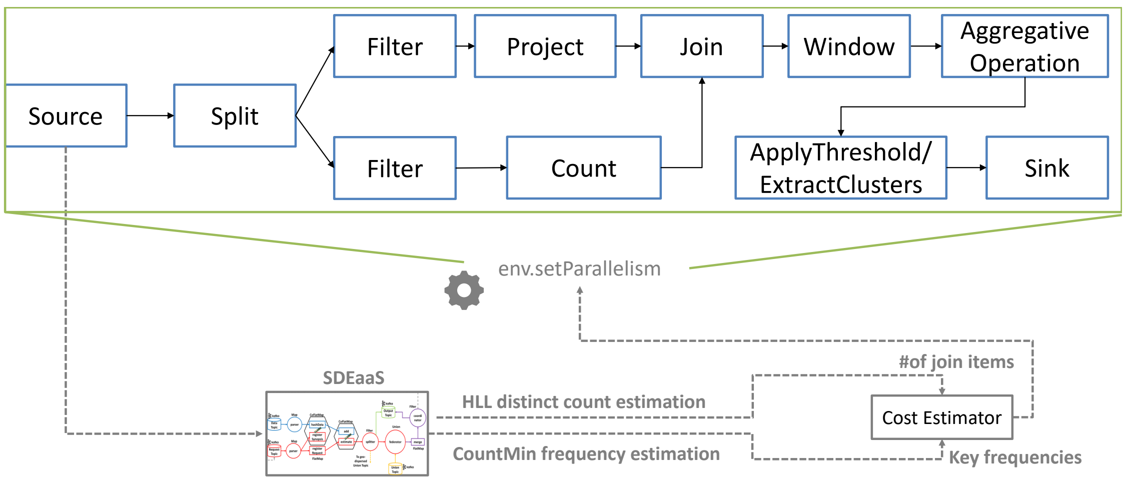


Figure 7: SDEaaS as a Cost Estimator for Enhanced Horizontal Scalability.

far (in the financial dataset), with probability at least $1 - \delta$. CountMin sketches are easily mergeable by adding up the corresponding arrays.

An intrinsic optimizer can use SDEaaS as the cost estimator and derive the cardinality of the set of stocks that need to be monitored per time unit by querying the HLL sketch. Moreover, the CountMin sketch can be queried for estimating the frequency of each stock. Based on the HLL estimation the optimizer knows how many pieces of work need to be assigned to the workers. And based on the frequency of each stock, the size of each piece of work is also known. Therefore, the optimizer can configure the number of workers and balance the load among them. The whole rationale is depicted in Figure 7. Horizontal scalability is enhanced compared to what is provided by the Big Data platform alone. This is due to having a priori (provided by the SDEaaS nature of the engine) adequate statistics to ensure that no worker is overloaded causing reduction in the overall throughput during the execution of the workflow.

SDEaaS for Locality-aware Hashing & Vertical Scalability. Consider that the `AggregativeOperation` in Figure 6 involves computing pairwise similarities of stock bid count time series based on Pearson's Correlation Coefficient. As discussed in Section 1, tracking the full correlation matrix results in a quadratic explosion in space and time which is simply infeasible for very large number of monitored stocks. Let us now see how the DFT synopsis (Table 1) can be used for performing locality-aware hashing of streams to buckets, assign buckets including time series of stocks to workers and prune the number of pairwise comparisons for time series that are not hashed nearby. For that purpose, the SDE should be queried in-between the `Window` and `AggregativeOperation` of Figure 6 so as to get the BucketID per stock, i.e., the id of the worker where the `AggregativeOperation` (pairwise similarity estimation) will be performed independently.

Our Discrete Fourier Transform (DFT)-based correlation estimation implementation is based on StatStream [5]. An important observation for assigning time series to buckets is that there is a direct relation between Pearson's correlation coefficient (denoted $Corr$ below) 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 DFT transforms a sequence of n (potentially complex) numbers $x_0 \dots, x_{n-1}$ into another sequence of complex numbers $X_0, \dots, X_{n-1}$, which is defined by the DFT coefficients, calculated as $X_F = \frac{1}{n}\sum_{k=1}^{(n-1)} x_k e^{\frac{i2\pi kF}{n}}$, for $F = 0, \dots, n-1$ and $i = \sqrt{-1}$.

Compression is achieved by restricting F in the above formula to few coefficients. There are a couple of additional properties of the DFT which are taken into consideration for parallelizing the processing load of pairwise comparisons among time series:

1. 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.

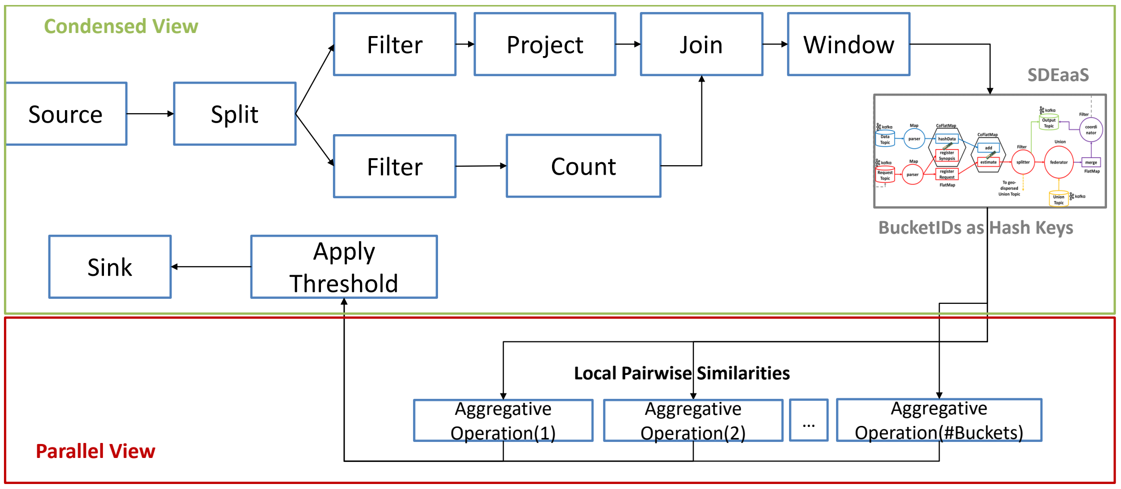


Figure 8: SDEaaS for Locality-aware Hashing & Vertical Scalability. The lower part provides a parallel view to explicitly depict that stock streams are hashed to different workers to compute pairwise similarities locally (independently), in parallel. Similarities among stocks streams are pruned for stocks that are not hashed nearby in the hashing space. The upper part of the figure shows a condensed view of the workflow (no parallel instances of operators are shown) to improve readability, but all operators (such as `ApplyThreshold`) can run in parallel instances.

2. It holds that $Corr(x, y) \geq 1 - \epsilon^2 \Rightarrow d(X', Y') \leq \epsilon$. This says that it is meaningful to examine only pairs of time series for which $d(X', Y') \leq \epsilon$. 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 workers.

The DFT coefficients can be updated incrementally upon operating over sliding windows [5]. 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 neighboring buckets, while the rest are hashed to distant buckets and, therefore, they are never compared for similarity. Time series that are hashed to more than one buckets are replicated an equal amount of times.

Now, assume a user-defined threshold T. 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 $T = 1 - \epsilon^2$. By using the DFT on normalized series, the original series 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 $\sqrt{2}/2$.

Based on the above observation, [5] notices that the range of each DFT coefficient is between $-\sqrt{2}/2$ and $\sqrt{2}/2$. 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 structure, composed of buckets for hashing groups of time series to each of them. Each bucket in the grid is of diameter ϵ and there are in total $2\lceil \frac{\sqrt{2}}{2\epsilon} \rceil^{(\#used\_coefficients)}$ buckets. For instance, in [5] 16-40 DFT coefficients are used to approximate stock exchange time series.

Each time series is hashed to a specific bucket inside the grid. Suppose X' is hashed to a bucket. To detect the time series whose correlation with X' is above T, only time series hashed to the same or adjacent buckets are possible candidates. Those time series are a super-set of the true set of highly-correlated ones. Since the bucket diameter is ϵ, time series mapped to non-adjacent buckets 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 similarity checks that are pruned while their score would pass the threshold.

Again, note that here the principal role of the SDEaaS is to produce the corresponding DFT coefficients and hash time series to buckets. That is why it should be queried between the `Window` and the `AggregativeOperation`. Therefore, the output of the corresponding synopsis in Table 1 includes the resulted coefficients and the bucket identifier. The actual similarity tests (in each bucket) may be performed by the downstream operator (`AggregativeOperation`) using the original time series. Figure 8 illustrates the partitioning rationale where SDEaaS and DFTs are used to produce the BucketID for each time series which are grouped/hashed to parallel instances of the `AggregativeOperation` of our case study, for similarity comparison.

SDEaaS for Synopsis-based Optimization & Enhanced Horizontal Scalability. When an application is willing to bargain accuracy for a considerable processing speed up or reduced memory consumption, the SDEaaS can act as the main tool of an advanced optimizer which would receive the application's accuracy budget and rewrite the workflow to equivalent but approximate forms so as to achieve the aforementioned performance goals.

Consider the workflow of Figure 6. Since CountMin sketches are not preferable for correlation estimation [7] in our discussion we are going to engage AMS sketches [32]. The key idea in AMS sketches 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 pseudo-random 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 arrives, $v[k] \cdot \xi_i[k]$ is added to the aforementioned sum and similarly for element deletion. Each sketch vector can be viewed as a two-dimensional $w \times d$ array, where $w = O(1/\epsilon^2)$ and $d = O(log(1/\delta))$, with ϵ, $1 - \delta$ being the desired bounds on error and probabilistic confidence, correspondingly. The inner product in the sketch-vector space and the L_2 norms (in which case we replace $sk(v_2)$ with $sk(v_1)$ in the formula below and vice versa) is defined as: $sk(v_1) \cdot sk(v_2) = \underbrace{median}_{j=1..d}\left\{\frac{1}{w} \sum_{i=1}^{w} sk(v_1)[i, j] \cdot sk(v_2)[i, j]\right\}$.

Some workflow execution plans that can be produced using our SDEaaS functionality and an accuracy budget are:

Plan1 The `Count` operator in Figure 6 can be rewritten to a `SDE.AMS` (sketches) operator to provide count (point query [43]) estimations of involved stocks and then use these sketches to judge pairwise similarities in `Aggregative Operation`.

Plan2 The `SDE.DFT` synopsis can replace the `Window` and `AggregativeOperation` operators to: (i) bucketize time series comparisons, (ii) speed up similarity tests by approximating original time series with few DFT coefficients.

Plan3 Rewrite the `Count` operator to `SDE.AMS` and rewrite the `Window` and `AggregativeOperation` to `SDE.DFT` in which case the DFT operates on the sketched instead of the original time series. This transformed workflow is what we present in Figure 5.

Based on which plans abide by the accuracy budget and on the time and space complexity guarantees of each synopsis, the optimizer can pick the workflow execution plan that is expected to provide the higher throughput or lower memory usage. Again, horizontal scalability is enhanced compared to what the Big Data platform alone provides, by using the potential of synopses.

SDEaaS for AQP & Federated Scalability. In the scope of Approximate Query Processing (AQP), the workflow of Figure 6 can take advantage of federated synopses that are supported by our SDEaaS architecture (Figure 2, Section 4.2) in order to reduce the amount of data that are communicated and, thus, to enable federated scalability. For instance, assume Level 1, Level 2 data of stocks first arrive at sites (computer clusters each running our SDEaaS) located at the various countries of the corresponding stock markets. Should one wish to pinpoint correlations of stocks globally, a need to communicate the windowed time series of Figure 6 occurs. To ensure federated scalability to geo-dispersed settings composed of many sites, only few coefficients of `SDE.DFT` or `SDE.AMS` sketches can be used to replace the `Window` operator in Figure 6 and reduce the dimensionality of the time series. Hence, the communication cost, because compressed time series are exchanged among the sites, is harnessed and network latencies are prevented.

Figure 9 illustrates the above rationale using 3 sites. Among these, Site 1 is set as the responsible one for synthesizing the synopses of all the three sites. SDEaaS instances at Site 2, Site 3 communicate only the locally computed $< BucketID, DFTCoefficients >$ which constitute dimensionality reduced versions of their local time series. Having done that, they act as producers to the `UnionTopic` of SDEaaS running at Site 1. Site 1 merges all partial synopses synopses and provides them to the `OutputTopic` at Site 1, where they are consumed by the downstream operator `AggregativeOperation`. Notice that DFT coefficients are used by the `AggregativeOperation`, instead of the original time series. This time we not only bucketize time series as in Figure 8, but also we approximate them and estimate their similarity score using few DFT coefficients.

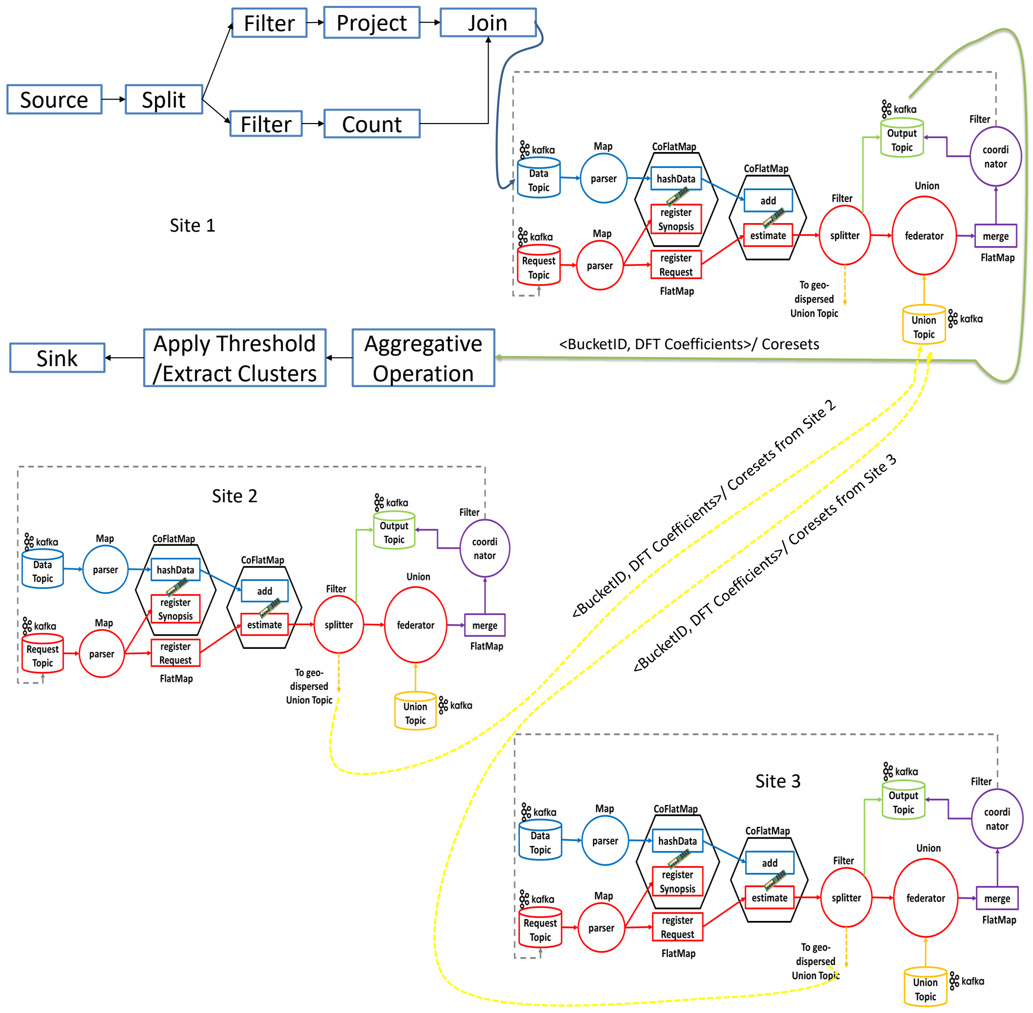


Figure 9: SDEaaS for AQP & Federated Scalability and for Online Data Stream Mining. The workflows of Site 2 and Site 3 are not depicted to improve readability. Few coefficients of `SDE.DFT` or `SDE.CoreSetTree` are transmitted and thus synopses reduce the communication cost.

SDEaaS for Online Data Stream Mining. StreamKM++ [33] is a streaming clustering algorithm that bases its function on a carefully-crafted sample of a data stream, called CoreSet. A CoreSet is a small weighted subset of the original stream that can be used to approximate the solution to the clustering problem, with certain quality guarantees. A data structure termed CoreSetTree is used to speed up the time necessary for sampling non-uniformly during CoreSet construction and maintenance. The CoreSetTree is a binary tree which describes a hierarchical structure, where the root represents the whole set of points P in the data stream and the children of each node p in the tree represent a partition of the elements. After the CoreSet is extracted from a data stream, a weighted k-means algorithm is applied to get the final clusters.

This synopsis is also supported by our SDE to provide the potential for online data stream mining, clustering - in particular, purposes. In the case of stock time series clustering, the `AggregativeOperation` in Figure 6 is to be replaced by `SDE.CoreSetTree` and the `ExtractClusters` operator is a weighted k-means that uses the CoreSets (Figure 9).

SDEaaS on Volatile-energy Streams. Stakeholders are often not interested in the correlations hidden directly in stock price streams, but their focus is rather on the analysis of stock returns instead. Stock returns constitute fractional changes of stock prices from one timepoint to another. Contrary to stock price time series which concentrate most of

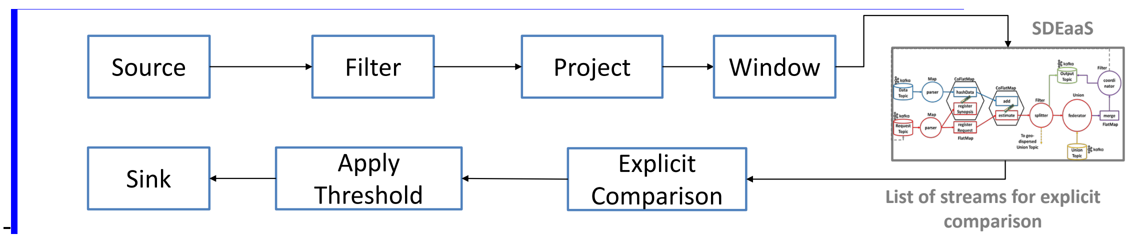


Figure 10: Workflow for Analyzing Stock Returns (Condensed View).

their energy in few SDE.DFT coefficients, stock return analysis is a much more challenging task because stock returns maintain power in all frequencies and cannot be accurately summarized by a few Fourier coefficients.

To support such cases the SDE implements and incorporates in its algorithmic arsenal the line of work introduced in [13, 14] (see Table 1). We term this line of work as the `SDE.RadiusSketch` family. `SDE.RadiusSketch` first applies LSH on each windowed stream to produce sketch summaries of the original streams. Based on these summaries, it constructs a number of grids each containing the sketch values corresponding to a subset of LSH random vectors. Fragments (subvectors) of each original stream window are hashed to different grids maintained by various workers, based on their LSH signature. A pair of streams is candidate to be explicitly compared for similarity if the number of subvectors that are assigned to the same grid cell across all grids, exceeds a given threshold. Otherwise, explicit comparison is avoided.

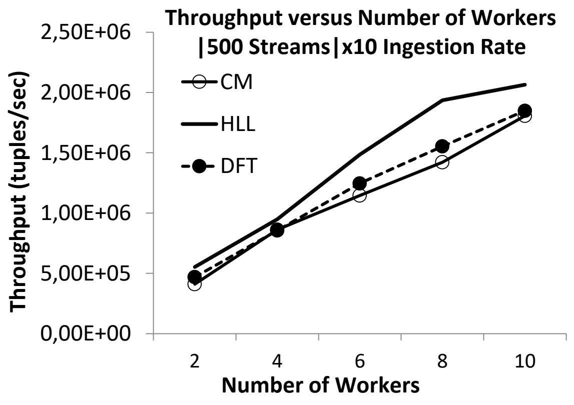
In Figure 10, stock trades arrive at the `Source` and are optionally Filtered to monitor only a subset of stocks. The `Project` operator keeps the timestamp of the trade and computes stock returns. The corresponding result is inserted in a `Window` of recent return values, forming a stock return time series. SDEaaS utilizes `SDE.RadiusSketch` to extract a list of streams that should be explicitly compared for correlation/similarity. Finally, the results in the form of pairs of stocks surpassing a similarity threshold (`ApplyThreshold` operator in Figure 10) are directed to a `Sink`.

The challenging part that may stress-test the performance of our SDEaaS approach is the fragmentation of the streams to subvectors and the partitioning to multiple grids. This is because the kind of partitioning introduced by `SDE.RadiusSketch` essentially poses a computational barrier inside the SDE: the decision about explicit (or not) similarity computation is based on aggregating the preliminary similatirity tests of subvectors fragmented to multiple workers. Therefore, the decision cannot be made unless all workers are done with the work assigned to them. The fundamental difference with `SDE.DFT` is that `SDE.DFT` uses few Fourier coefficients to hash entire (non-fragmented) streams to workers. Each worker, then computes independently the similarity of pairs of streams that are hashed to it. Therefore, though inaccurate and, therefore inapplicable for streams that maintain energy across all frequencies, `SDE.DFT` does not pose a computational barrier.

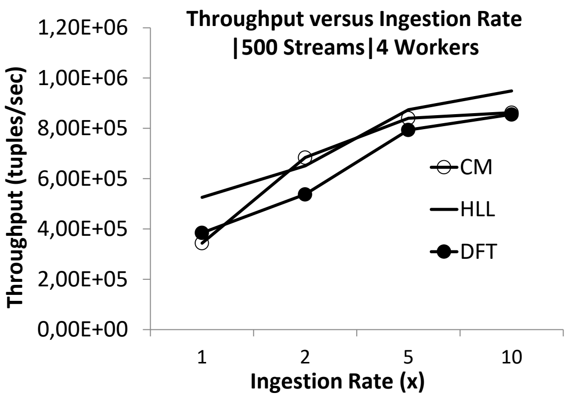
Our experimental evaluation in Section 8 and Figure 14 illustrates that even in such challenging scenarios SDEaaS with the use of `SDE.RadiusSketch` can provide up to 7.5 times performance improvement compared to the second best alternative.

8. Experimental Evaluation

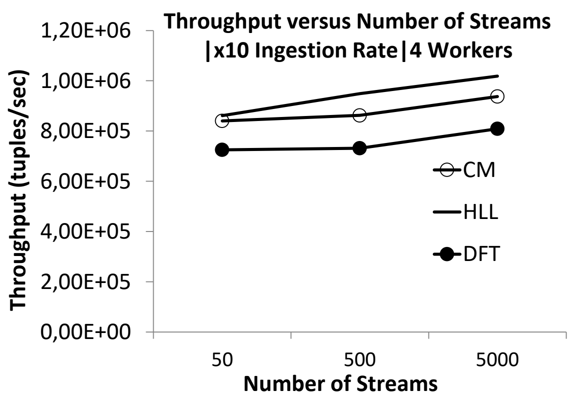
The SDE has been implemented in about 10K lines of code in the Java DataStream API of Flink [39]. To test the performance of our SDEaaS approach, we utilize a Kafka cluster with 3 Dell PowerEdge R320 Intel Xeon E5-2430 v2 2.50GHz machines with 32GB RAM each and one Dell PowerEdge R310 Quad Core Xeon X3440 2.53GHz machine with 16GB RAM. Our Flink cluster has 10 Dell PowerEdge R300 Quad Core Xeon X3323 2.5GHz machines with 8GB RAM each. We use a real dataset composed of ~5000 stocks contributing a total of ~10 TB of Level 1 and Level 2 data. Part of these data are available open-source [44, 45]. 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 related works cited in Table 1.



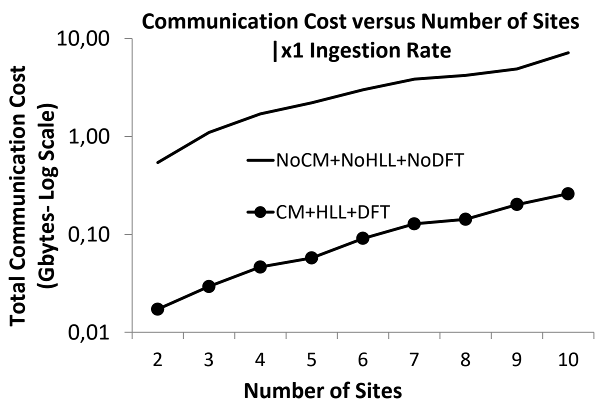
(a) Varying the Parallelism.



(b) Varying the Ingestion Rate.



(c) Varying the Number of Streams.



(d) Varying the Number of Sites.

Figure 11: SDEaaS Scalability Study.

8.1. 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 operators, without placing these operators provided by the SDE as parts of a workflow. In particular, we measure the throughput, expressed as the number of tuples being processed per time unit (second) and communication cost (Gbytes) among workers, while varying a number of parameters involving horizontal ((i),(ii)), vertical (iii) and federated (iv) scalability, respectively: (i) the parallelization degree [2-4-6-8-10], (ii) 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), (iii) the number of summarized stocks (streams) [50-500-5000] and (iv) the Gbytes 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, $\epsilon = 0.002, \delta = 0.01$), AMS ($\epsilon = 0.002, \delta = 0.01$) synopses each of which, as discussed in Section 7, is destined to support different types of analytics related to correlation, distinct count and frequency estimation, respectively (Table 1). Since the CM and the AMS sketches exhibited very similar performance we only include CM sketches in the graph to improve readability. 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.

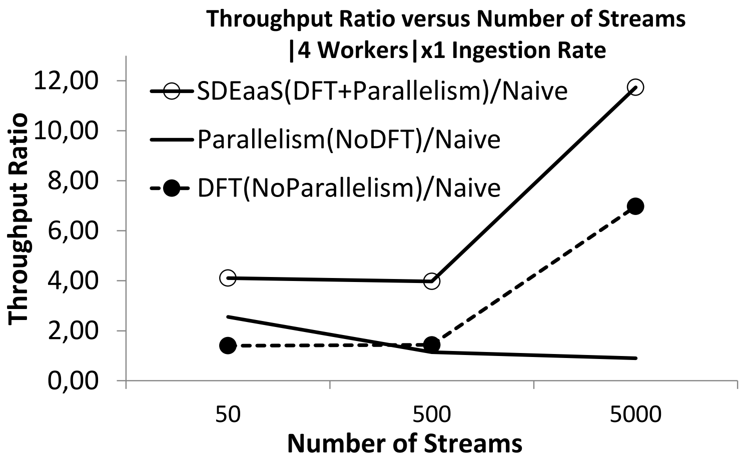


Figure 12: Comparative Analysis in Executing the Workflow of Figure 6 using `SDE.DFT`.

Figure 11(a) 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 11(b), 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 11(c) 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. We further comment on such aspects in the comparative analysis in Section 8.2.

Finally, Figure 11(d) illustrates the communication performance of SDEaaS upon maintaining federated synopses and communicating the results to a responsible site so as to derive the final estimations (see yellow arrows in Figure 2 and Section 4.2). For this experiment, we divide the streams among workers and each worker represents a site which analyzes 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 5 minutes.

Here, the total communication cost for deriving estimations from synopses, is not a number that says much on its own. It is expected for the communication cost to 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 11(d)) compared to when we do not. Therefore, in Figure 11(d), we also plot NoCM+NoHLL+NoDFT illustrating the communication cost that takes place upon answering the same (cardinality, count, time series) queries without synopses. As Figure 11(d) illustrates (the vertical axis is in log scale), the communication gains steadily remain above an order of magnitude.

8.2. Comparison against Parallel and Sketching Candidates

We use the DFT synopsis to replace `Window`, `AggregativeOperation` as discussed in Section 7, since the most computationally intensive (and thus candidate to become the bottleneck) operator in the workflow of Figure 6 is the `AggregativeOperation` which performs pairwise correlation estimations of time series. Indicatively, when 5K stocks are monitored, the pairwise similarity comparisons that need to be performed by naive approaches are 12.5M.

In Figure 12 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 12) 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. Its performance corresponds to competitors such as [25, 27, 26] which provide facilities for parallel synopses maintenance, but the utilized synopses are deprived from vertical scalability features.
- **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 of Figure 6. 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 [23] or Stream-lib [24] 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 12 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 Naive approach. On the other hand, the Parallelism(NoDFT) improves over the Naive by ~2.5 times. Our SDEaaS(DFT+Parallelism), taking advantage of both the synopsis and parallelism improves over the Naive by almost 4 times. Note that when 50 streams are monitored, the number of performed pair-wise similarity checks in the workflow of Figure 6 for the Naive approach is 2.5K/2.

This is important because, according to Figure 12, 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 `AggregativeOperation` starting to become a computational bottleneck for Parallelism(NoDFT) in the workflow of Figure 6. 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 12 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 12). This indicates the success of SDEaaS in integrating the virtues of data synopsis and parallel processing.

We then perform a similar experiment for the stream mining version of the workflow in Figure 6 as described in Section 7. In particular, in this experiment the Naive approach corresponds to StreamKM++ clustering without parallelism and coreset sizes equivalent to the original data points (time series). The Parallelism(NoCoreSetTree) approach involves performing StreamKM++ with coreset sizes equivalent to the original data points, but exploiting parallelism. The CoreSetTree(NoParallelism) exploits the CoreSetTree synopsis but uses no parallelism, while SDEaaS(CoreSetTree + Parallelism) combines the two. For CoreSetTree(NoParallelism) and SDEaaS(CoreSetTree +

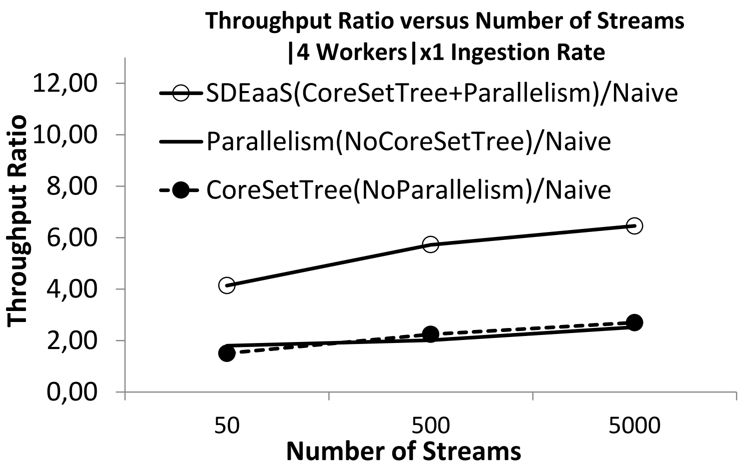


Figure 13: Comparative Analysis in Executing the Workflow of Figure 6 using `SDE.CoreSetTree`.

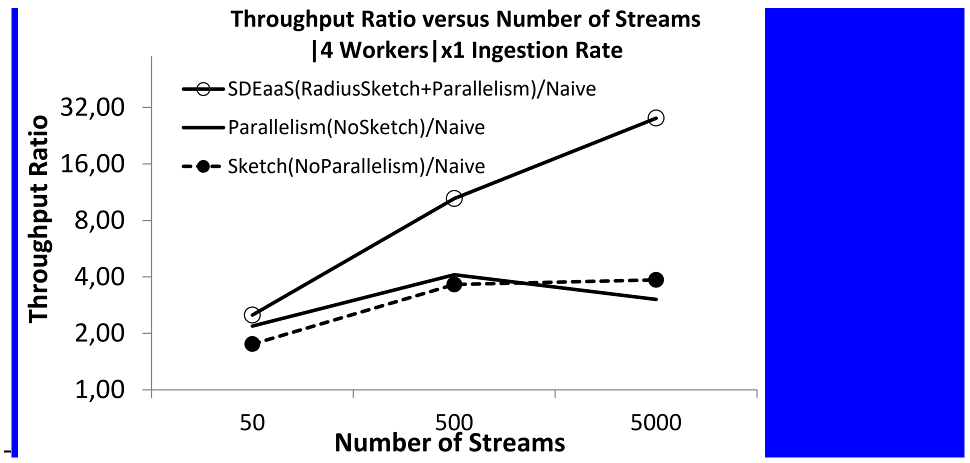


Figure 14: Comparative Analysis Executing the Workflow of Figure 10 on Stock Returns using the `SDE.RadiusSketch` Family [13, 14].

Parallelism), we use bucket sizes of 10-100-400 and k values are set to $4 - 10 - 40$, for 50-500-5000 streams, correspondingly. The conclusions that can be drawn from Figure 13 are very similar with what we discussed in Figure 12. However, the respective ratios of throughput over the Naive approach are lower (2-3 times higher throughput than the second best candidate in Figure 13). This is by design of the mining algorithm and the reason is that the clustering procedure includes a reduction step which cannot be executed in parallel and is, thus, performed by a single worker. This is in contrast with the `ApplyThreshold` operation in Figure 12 which can be performed in parallel by different processing units, independently.

8.3. Stress-testing SDEaaS on Volatile-energy Streams

In Section 7 we argued about why analyzing stock returns poses special challenges to our SDEaaS approach. To stress-test the computational performance of the SDE in such challenging scenarios we execute the workflow of Figure 10 concentrating on returns computed over windows of size 256. We parameterize the Radius Sketch synopsis according to related work [13].

Figure 14 illustrates the performance of our SDEaaS approach against other competitors. As discussed in Section 7, SDEaaS applies Radius Sketch [13, 14] to prune the number of explicit stream comparisons. The line labeled

SDEaaS(RadiusSketch+Parallelism)/Naive in Figure 14 shows the improvement of our approach over the Naive one. Again, the Naive approach corresponds to calculating stock return time series' distance without exploiting synopses or parallel processing. The Sketch(NoParallelism)/Naive line in the figure exhibits the performance gains over the Naive approach when we use the LSH skecthes to reduce the dimensionality of the original streams. In particular, Sketch(NoParallelism) performs stream comparisons using LSH sketches, but without exploiting parallelism or locality-aware hashing for comparison pruning. Finally, Parallelism(NoSketch)/Naive reflects performance gains from parallel distance computation without the vertical scalability provided by the Radius Sketch synopsis.

In Figure 14 we observe that for a small number of 50 streams all approaches perform almost equivalently, improving over Naive by 1.75 for Sketch(NoParallelism) and up to 2.5 times for SDEaaS(RadiusSketch+Parallelism). When we switch to 500 streams, Sketch(NoParallelism) improves over Naive by 3.7 times, while the improvement provided by Parallelism(NoSketch) is 4.2 times. SDEaaS(RadiusSketch+Parallelism) improves over Naive by an order of magnitude and, if we compare the lines of SDEaaS(RadiusSketch+Parallelism)/Naive with Parallelism(NoSketch)/Naive, we can see that SDEaaS improves the second best approach by 2.6 times.

Upon switching to 5000 streams, the performance of Parallelism(NoSketch) becomes worse compared to Sketch(NoParallelism) validating once again the observation we made in Section 8.2: the mere use of parallelism is not sufficient performance-wise because it lacks the potential for vertical scalability. SDEaaS(RadiusSketch+Parallelism) improves the second best Sketch(NoParallelism) approach by approximately 7.5 times and also improves Parallelism(NoSketch) by almost 10 times. The latter factor of $\times 10$ is due to comparison pruning introduced by the employed Radius Sketch, since both SDEaaS(RadiusSketch+Parallelism) and Parallelism(NoSketch) operate over 4 worker nodes.

The question that needs to be answered involves the reason why SDEaaS(RadiusSketch+Parallelism) improves the Naive approach by $\sim$28 times as shown by the respective plot line. Notice that, for 5000 streams, Parallelism(NoSketch) improves the Naive approach by $\sim$ 3 times, as shown in Figure 14. This improvement is purely due to parallelism. On the other hand, we just saw that SDEaaS(RadiusSketch+Parallelism) improves Parallelism(NoSketch) by an order of magnitude due to comparison pruning. Roughly speaking, the throughput ratio of SDEaaS(RadiusSketch+Parallelism) is $\sim$28 times because it causes a $\times \sim 10$ factor improvement to the throughput of Naive due to explicit comparison pruning and another $\times \sim 3$ factor due to the employed parallelism in the remaining explicit comparisons.

Note that the $\times$ 3 factor of Parallelism(NoSketch) and sub-factor of SDEaaS(RadiusSketch+Parallelism) are reduced compared to the employed parallelization scheme which uses 4 workers in this experiment. This reduction is attributed to the barrier discussed in Section 7. Still, the performance of SDEaaS, upon leveraging the RadiusSketch family, is starking for high number of streams against all other competitors. Finally, we again stress that SDEaaS approach does not alter in anyway the quality (accuracy) of Radius Sketch as extensively reviewed in related work [13, 14].

8.4. SDEaaS vs non-SDEaaS Summarization Approaches

In Section 6 we argued about the fact that employing a non-SDEaaS approach, as works such as [27, 25, 26] do, restricts the maximum allowed number of concurrently maintained synopses up to the available task slots. That is, if the SDE is not provided as a service using our novel architecture, in case we want to maintain a new synopsis when a demand arises (without ceasing the currently maintained ones, because these may already serve workflows as the one in Figure 6), we have to submit a new job. A job occupies at least one task slot. On the contrary, in our SDEaaS approach, when a request for a new synopsis arrives on-the-fly, we simply devote more tasks (which can exploit hyper-threading, pseudo-parallelism) instead of entire task slots. Because of that, our SDEaaS design is a much more preferable choice since it can simultaneously maintain thousands of synopses for thousands of streams.

To show the superiority of our approach in practice, we design an experiment where we start with maintaining 2 CM sketches for frequency estimations on the volume, price pairs of each stock. Note that this differs compared to what we did in Figure 11 where we kept a CM sketch for estimating the count of bids per stock in the whole dataset. Then, we express demands for maintaining one more CM sketch for up to 5000 sketches/stocks. We do that without stopping the already running synopses each time. We measure the sum of throughputs of all running jobs for the non-SDEaaS approach and the throughput of our SDE and plot the results in Figure 15.

First, it can be observed that non-SDEaaS cannot maintain more than 40 synopses simultaneously using the non-SDEaaS approach since it depletes the 40 available task slots. This is denoted with ✘ signs in the plot. Second, even when up to 40 synopses are concurrently maintained, our SDEaaS approach always performs better compared to the

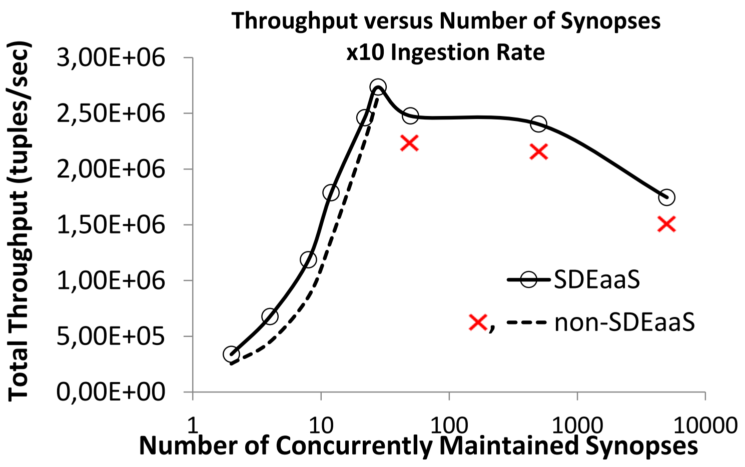


Figure 15: Comparison of SDAaaS vs non-SDEaaS. ✗ signs denote that non-SDEaaS cannot maintain more than 40 synopses simultaneously since available task slots are depleted.

non-SDEaaS alternative. This is because slot sharing in SDEaaS means that more than one task is scheduled into the same slot, or in other words, CM sketches end up sharing resources. The main benefit of this is better resource utilization. In the non-SDEaaS approach if there is skew in the update rate of a number of streams (to which one task slot per synopsis per stream is alloted), we might easily end up with some slots doing very little work at certain intervals, while others are quite busy. This is avoided in SDEaaS due to slot sharing. Therefore, better resource utilization is an additional advantage of our SDEaaS approach towards scaling with high numbers of streams.

9. Conclusions and 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 endeavor. Our future work focuses on (a) enriching the SDE Library with more synopsis techniques [9], (b) integrate it with machine- and deep-learning frameworks [46], (c) implement the proposed SDEaaS architecture on Apache Beam [47], to make the service directly runnable to a variety of Big Data platforms.

Acknowledgment

Antonios Kontaxakis and Dimitris Sacharidis were supported by the European Union under Horizon 2020 grant agreement No. 955895. Nikos Giatrakos was supported by the EU project EVENFLOW under Horizon Europe agreement No. 101070430. Antonios Deligiannakis was supported by the EU project STELAR Horizon Europe - Grant No. 101070122.

References

[1] Forbes, https://www.forbes.com/sites/tomgroenfeldt/2013/02/14/at-nyse-the-data-deluge-overwhelms-traditional-databases/#362df2415aab.
[2] A. Milios, K. Bereta, K. Chatzikokolakis, D. Zissis, S. Matwin, Automatic fusion of satellite imagery and AIS data for vessel detection, in: 22th International Conference on Information Fusion, FUSION 2019, Ottawa, ON, Canada, July 2-5, 2019, 2019, pp. 1–5.
[3] E. Zeitler, T. Risch, Massive scale-out of expensive continuous queries, Proc. VLDB Endow. 4 (11) (2011) 1181–1188.
[4] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, V. Markl, Benchmarking distributed stream data processing systems, in: 34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018, 2018, pp. 1507–1518.

[5] Y. Zhu, D. E. Shasha, Statstream: Statistical monitoring of thousands of data streams in real time, in: Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002, 2002, pp. 358–369.

[6] G. S. Manku, R. Motwani, Approximate frequency counts over data streams, in: Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002, 2002, pp. 346–357.

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

[8] P. Flajolet, É. Fusy, O. Gandouet, F. Meunier, Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm, in: Discrete Mathematics and Theoretical Computer Science, Discrete Mathematics and Theoretical Computer Science, 2007, pp. 137–156.

[9] G. Cormode, M. Garofalakis, P. Haas, C. Jermaine, Synopses for massive data: Samples, histograms, wavelets, sketches, Foundations and Trends in Databases 4 (1-3) (2012) 1–294.

[10] Data stream management - processing high-speed data streams, Data-Centric Systems and Applications, Springer, 2016. `doi:10.1007/978-3-540-28608-0`.
URL `https://doi.org/10.1007/978-3-540-28608-0`

[11] G. Cormode, K. Yi, Small Summaries for Big Data, Cambridge University Press, 2020.

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

[13] Djamel Edine Yagoubi and Reza Akbarinia and Florent Masseglia and Dennis E. Shasha, RadiusSketch: Massively Distributed Indexing of Time Series, in: 2017 IEEE International Conference on Data Science and Advanced Analytics, DSAA 2017, Tokyo, Japan, October 19-21, 2017, IEEE, 2017, pp. 262–271.

[14] Oleksandra Levchenko and Boyan Kolev and Djamel Edine Yagoubi and Reza Akbarinia and Florent Masseglia and Themis Palpanas and Dennis E. Shasha and Patrick Valduriez, BestNeighbor: efficient evaluation of kNN queries on large time series databases, Knowl. Inf. Syst. 63 (2) (2021) 349–378.

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

[16] N. Giatrakos, A. Deligiannakis, M. N. Garofalakis, Y. Kotidis, Omnibus outlier detection in sensor networks using windowed locality sensitive hashing, Future Gener. Comput. Syst. 110 (2020) 587–609.

[17] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, K. Yi, Mergeable summaries, in: M. Benedikt, M. Krötzsch, M. Lenzerini (Eds.), Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012, ACM, 2012, pp. 23–34.

[18] A. Kontaxakis, N. Giatrakos, A. Deligiannakis, A synopses data engine for interactive extreme-scale analytics, in: CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020, 2020, pp. 2085–2088.

[19] Rapidminer studio, streaming extension, `https://marketplace.rapidminer.com/UpdateServer/faces/product_details.xhtml?productId=rmx_streaming`.

[20] SDEaaS v1.0.1, `https://sdeaas.github.io` (2023).

[21] M. Vodas, K. Bereta, D. Kladis, D. Zissis, E. Alevizos, E. Ntoulias, A. Artikis, A. Deligiannakis, A. Kontaxakis, N. Giatrakos, D. Arnu, E. Yaqub, F. Temme, M. Torok, R. Klinkenberg, Online distributed maritime event detection & forecasting over big vessel tracking data, in: 2021 IEEE International Conference on Big Data (Big Data), Orlando, FL, USA, December 15-18, 2021, 2021, pp. 2052–2057.

[22] G. Stamatakis, A. Kontaxakis, A. Simitsis, N. Giatrakos, A. Deligiannakis, Sheermp: Optimized streaming analytics-as-a-service over multi-site and multi-platform settings, in: Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022, 2022, pp. 2:558–2:561.

[23] Apache datasketches, `https://datasketches.github.io/`.

[24] Stream-lib, `https://github.com/addthis/stream-lib`.

[25] B. Mozafari, Snappydata, in: Encyclopedia of Big Data Technologies, 2019.

[26] D. L. Quoc, R. Chen, P. Bhatotia, C. Fetzer, V. Hilt, T. Strufe, Streamapprox: approximate computing for stream analytics, in: Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Las Vegas, NV, USA, December 11 - 15, 2017, 2017, pp. 185–197.

[27] R. P. Lemaitre, M. Kiefer, J. V. Hein, J. Quiané-Ruiz, V. Markl, In the land of data streams where synopses are missing, one framework to bring them all, Proc. VLDB Endow. 14 (10) (2021) 1818–1831.

[28] G. Cormode, S. Muthukrishnan, K. Yi, Q. Zhang, Optimal sampling from distributed streams, in: Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA, 2010, pp. 77–86.

[29] B. Babcock, M. Datar, R. Motwani, Sampling from a moving window over streaming data, in: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA, 2002, pp. 633–634.

[30] P. Flajolet, G. N. Martin, Probabilistic counting algorithms for data base applications, J. Comput. Syst. Sci. 31 (2) (1985) 182–209.

[31] B. H. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (7) (1970) 422–426.

[32] N. Alon, Y. Matias, M. Szegedy, The space complexity of approximating the frequency moments, in: Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996, 1996, pp. 20–29.

[33] M. R. Ackermann, M. Märtens, C. Raupach, K. Swierkot, C. Lammersen, C. Sohler, Streamkm++: A clustering algorithm for data streams, ACM J. Exp. Algorithmics 17 (1) (2012).

[34] M. Charikar, Similarity estimation techniques from rounding algorithms, in: Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada, 2002, pp. 380–388.

[35] M. Greenwald, S. Khanna, Space-efficient online computation of quantile summaries, in: S. Mehrotra, T. K. Sellis (Eds.), Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001, ACM, 2001, pp. 58–66.

[36] A. Arasu, G. S. Manku, Approximate counts and quantiles over sliding windows, in: C. Beeri, A. Deutsch (Eds.), Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France, ACM, 2004, pp. 286–296.

[37] S. Chintapalli, D. Dagit, B. Evans, et al, Benchmarking streaming computation engines: Storm, flink and spark streaming, in: IPDPS Workshops, 2016.
[38] Apache Kafka v. 3.3, `https://kafka.apache.org/`.
[39] Apache Flink v. 1.16, `https://flink.apache.org/`.
[40] J. Kreps, N. Narkhede, J. Rao, et al., Kafka: A distributed messaging system for log processing, in: Proceedings of the NetDB, 2011, pp. 1–7.
[41] N. Giatrakos, E. Alevizos, A. Artikis, A. Deligiannakis, M. N. Garofalakis, Complex event recognition in the big data era: a survey, VLDB J. 29 (1) (2020) 313–352.
[42] SDEaaS Client v1.0.1, `https://github.com/akontaxakis/SDE/tree/master/a-client-for-the-synopsis-data-engine` (2023).
[43] G. Cormode, M. N. Garofalakis, Join sizes, frequency moments, and applications, in: Data Stream Management - Processing High-Speed Data Streams, 2016, pp. 87–102.
[44] spring, Financial data set used in infore project (Jun. 2020). `doi:10.5281/zenodo.3886895`.
URL `https://doi.org/10.5281/zenodo.3886895`
[45] Burkard, Data set of correlations between stocks world wide (Mar. 2022). `doi:10.5281/zenodo.6331464`.
URL `https://doi.org/10.5281/zenodo.6331464`
[46] Deep learning on flink, `https://github.com/flink-extended/dl-on-flink`.
[47] Apache Beam v. 2.41.0, `https://beam.apache.org/`.