



Data Synopses Generator V1
Work Package 6 Task 6.1 Deliverable 6.1

Authors

Nikos Giatrakos, Antonis Kontaxakis, Antonios Deligiannakis
Athena Research & Innovation Center

Marios Vodas
MarineTraffic

 Project supported by the European Commission Contract no. 825070	WP6 T6.1 Deliverable D6.1	Doc.nr.: WP6 D6.1
		Rev.: 1.0
		Date: 20/12/2019
		Class.: Public



Distribution list:

Groups:	Others:
WP Leader: NCSR Demokritos Task Leader: Athena	Internal Reviewer Partner: MarineTraffic (MT) INFORE Management Board INFORE Project Officer

Document history:

Revision	Date	Section	Page	Modification
0.1	25/11/2019	1-2	1-7	Creation
0.2	26/11/2019	3.1	7-12	Creation
0.3	27/11/2019	3.2	12-14	Creation
0.4	28/11/2019	4, 5.1, 5.2	15-23	Creation
0.5	29/11/2019	5	18-30	Self-review, extended with 5.3
0.6	03/12/2019	6,7,8	32-35	Creation
0.7	04/12/2019	All	All	Self-review, submitted for internal review
0.8	13/12/2019	All	All	Internal review completed
0.9	18/12/2019	All	All	Internal review comments incorporated
1.0	20/12/2019	All	All	Final review and comments incorporation

Approvals:

First Author: Nikos Giatrakos (Athena) Date: 04/12/2019

Internal Reviewer: K.Bereta, K. Chatzikokolakis (MT) Date: 13/12/2019

Coordinator: Antonios Deligiannakis (Athena) Date: 20/12/2019

 Project supported by the European Commission Contract no. 825070	WP6 T6.1 Deliverable D6.1	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public



Table of contents:

1	Executive Summary	4
2	Introduction.....	5
3	INFORE Requirements by the SDE Component	7
3.1	Non-functional Requirements – Implementation Platform.....	7
3.1.1	Support for Lambda and Kappa Architectures	7
3.1.2	Event Processing Orientation & Ease of Use, Flexibility, Reliability.....	9
3.1.3	Scalability Considerations.....	11
3.1.4	Pluggability Considerations	11
3.1.5	Other Desirable Features.....	12
3.2	Functional Requirements	12
3.2.1	Single-stream Synopsis Maintenance.....	13
3.2.2	Dataset Synopsis Maintenance	13
3.2.3	Multi-stream Synopsis Maintenance	13
3.2.4	Ad-hoc Querying Capabilities.....	13
3.2.5	Continuous Querying Capabilities	13
3.2.6	Dynamic Build/Stop of Synopses.....	13
3.2.7	Providing SDE Status Report	14
3.2.8	External Synopsis Load and Maintenance	14
4	SDE Architecture	15
4.1	Employed Parallelization Scheme(s).....	15
4.2	Data and Query Ingestion	16
4.3	Requesting New Synopsis Maintenance	16
4.4	Updating the Synopsis	16
4.5	Ad-hoc Query Answering.....	16
4.6	Continuous Query Answering	17
5	The Synopses Library	18
5.1	Structure of the Synopses Library.....	18
5.2	Supported Synopses Tied to INFORE Use Cases.....	18
5.2.1	Random Hyperplane Projection (RHP) Locality Sensitive Hashing (LSH).....	19
5.2.2	Discrete Fourier Transform (DFT).....	21
5.2.3	STSampler Synopsis	23
5.3	Supported Synopsis for Broader Application Scenarios	23
5.3.1	Lossy Counting	23
5.3.2	Sticky Sampling	25
5.3.3	CountMin Sketch	25
5.3.4	FM Sketch	26
5.3.5	Bloom Filters.....	26
5.3.6	HyperLogLog Sketch	27
5.3.7	AMS Sketch	28
5.3.8	Chain Sampling.....	28
5.3.9	GK Quantiles.....	29
6	Performance Evaluation.....	31
7	Conclusions and Future Work.....	33
8	References.....	34

 <p>Project supported by the European Commission Contract no. 825070</p>	<h2>WP6 T6.1</h2> <h3>Deliverable D6.1</h3>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public



1 Executive Summary

This deliverable elaborates on the first version of the Data Synopses Generator, henceforth referred to as Synopsis Data Engine (SDE), developed within the scope of INFORE. The purpose of the SDE is to provide concise summaries of the massive, high-velocity input data streams it processes, both in per stream as well as cross-stream fashion. The data summaries provided by the SDE constitute representative data views of important aspects (samples, expected values, counts, frequency moments, among others) of the incoming data, approximated with predefined accuracy guarantees. Such synopses that effectively summarize huge volumes of ever growing data in a real-time, online fashion can be used by other INFORE modules such as online learning, complex event forecasting algorithms in WP6 or the INFORE Optimizer developed in WP5 to effectively work on carefully crafted data summaries instead of the entire stream(s) and, speed up response times of complex tasks in INFORE workflows, thus boosting interactive data exploration.

Given the above, the current deliverable outlines the requirements posed by INFORE that led the strategic decisions behind SDE’s architectural design. It details the architectural choices in the development of the SDE, from the point streaming data are digested from relevant data sources to the point where concise synopses of these data are built, maintained and get subsequently delivered as responses to respective requests by other INFORE components. We further describe the functionality and the design of the currently supported synopses. To ensure the utility of the SDE in INFORE use case scenarios and broader application areas, the synopses currently incorporated in the SDE include general-purpose streaming data summarization techniques as well as synopses destined to support INFORE use case workflows.

For the reasons argued in this deliverable, the SDE has been developed on Apache Flink Big Data platform, one of the most prominent frameworks designed to operate as a true streaming engine and to combine batch and stream operations. The SDE is fully extensible as each new, specific synopsis algorithm can be plugged in by extending the Synopsis supertype and simply instantiating the basic methods. The first version of the SDE allows the on-the-fly (i.e., as the SDE is up and running): (a) deployment and maintenance of a new synopsis from a library of registered to the SDE algorithms for a given StreamID (e.g. stockID) or dataset (e.g. financial data¹), (b) querying in an ad-hoc or continuous fashion a maintained synopsis to provide a response with respect to the quantity it is destined to approximate, (c) querying all maintained synopses for a given <DatasetID, StreamID>, triplet of <DatasetID, StreamID, Value>, or for a given <DatasetID>, pair of <DatasetID, Value>, (d) providing reports on the currently running synopses and their parameters.

Within the scope of INFORE, the SDE may summarize data that directly originate from cancer evolution simulations, financial or maritime activity monitoring scenarios. Therefore, this deliverable accounts for scenarios described in Deliverables D1.1, D2.1 and D3.1, respectively, as the SDE is destined to support synopses tailored to the INFORE use cases besides being extensible to broader application areas. It further takes into consideration dataset descriptions outlined in the Data Management Plan V1, Deliverable D8.3. Besides, in the streaming workflows supported by INFORE, a synopsis provided by the SDE may act as an intermediate operation between other operators some of which provide input streams to be summarized (upstream operators), while the rest receiving the constructed summaries as their own input (downstream operators). Upstream and downstream operators may involve machine learning or forecasting operators developed in Tasks T6.2, T6.3 of WP6. Thus, the current deliverable directly interacts with the advancements made in these tasks, described in the upcoming Deliverable D6.2. In addition, INFORE’s Optimization module developed in WP5 should take into consideration that in case approximate query answers with predefined accuracy guarantees can be tolerated in a given workflow, combinations of operators in the workflow designed by the application may be substituted with approximate ones provided by the SDE. As such, the techniques developed in the upcoming versions of the Workflow Optimization Technology D5.2-D5.4, D5.7 can leverage SDE’s arsenal in their algorithms. Finally, the interactions of the SDE with other parts of the overall INFORE architecture are described in Deliverable D4.1.

¹ The term “dataset” throughout the deliverable is used to refer to a source of data that may report on the evolution of multiple streams. For instance, the source of the Financial use case of the project, reports on trades of multiple streams, one per stock.

 Project supported by the European Commission Contract no. 825070	<h2>WP6 T6.1</h2> <h3>Deliverable D6.1</h3>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public



2 Introduction

Interactive extreme-scale analytics over voluminous, high speed data streams become of the essence both within the scope of INFORE use cases and in a wide variety of broader, modern application scenarios. For instance, in the Life Science use case, studying the effect of applying combinations of drugs on simulated tumors of realistic sizes can generate cell state data of 100 GB/min [1], which need to be analyzed online to interactively determine successive drug combinations. Similarly, in the Financial use case, online discovery of stock correlations, useful for suggesting investment opportunities or predicting systemic risks, 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 produced at rates of hundreds of millions of tuples per second. Eventually, in the Maritime use case, complex events, in most cases, engage more than one vessel and corresponding trajectories, as for instance ship-to-ship transfer and piracy events. These also involve monitoring proximity and other relations among thousands of vessels within a specific spatiotemporal window.

A variety of Big Data platforms have been developed that support or are especially dedicated to stream processing. Such platforms aim at horizontal scalability, i.e., at scaling out the computation to a number of processing units available at a corporate data center (cluster) or cloud, by parallelizing the computational load and adaptively assigning computing resources to running analytics queries. However, relying only on these platforms might not be enough in order to provide interactive analytics at extreme scale. In the context of the Financial use case of INFORE, for instance, it is not straightforward to set threshold values in order to determine correlated pairs of stocks. To do so, one may need to run identical copies of an online Machine Learning algorithm under different parameters, in parallel. Such exploratory analysis may take hours before determining which model should be deployed to extract stock correlations and, until then, the specified thresholds might already become outdated. All the above compose a scenario in which interactivity is not facilitated by the chosen Big Data platform alone. Moreover, INFORE workflows may engage operators implemented in a variety of Big Data platforms.

On the other hand, data summarization techniques such as samples, sketches or histograms provide the fundamentals for enabling interactive analytics. They build carefully-crafted synopses of Big streaming Data which preserve data properties important for providing approximate answers, with tunable accuracy guarantees. Such queries include, but are not limited to, cardinality, frequency moment, correlation or quantile estimation. Since data summarization sheds the computational load, the complexity of the problem at hand is reduced and execution demanding tasks are severely sped up.

The goal and utility of the Synopses Data Engine (SDE) in INFORE is to combine the potential of data summarization techniques and the scalability primitives provided by Big Data platforms towards delivering interactive analytics at extreme scale. Our SDE is capable of providing:

- various types of scalability including:
 - enhanced horizontal scalability, i.e., further scaling the computation with the volume and velocity of Big streaming data,
 - vertical scalability, i.e., the ability to scale the computation with the number of monitored streams,
 - federated scalability, i.e., the ability to scale the computation in settings composed of multiple, potentially geographically dispersed clusters by exploiting data summarization to reduce communication.
- data summarization facilities even when used in application workflows engaging different Big Data platforms,
- pluggability and dynamic loading of new synopses on-the-fly,
- the primitives for being utilized by the INFORE Optimizer component so that it may replace exact operators of a workflow, with their approximate version to minimize workflow execution times under accuracy constraints.

In this deliverable we present the design principles and the structure of the SDE INFORE Component, we detail the developed SDE synopses library and we evaluate the first version of the SDE from a scalability viewpoint. The rest of the document is organized as follows: Section 3.1 presents the non-functional requirements that mainly drove our choice of developing the proof-of-concept SDE using Apache Flink and Kafka, in Java. Section 3.2 details the functional requirements that were taken into account during developing the SDE, which essentially constitute the functionality the SDE can provide to other operators in a given workflow. In Section 4 we present the SDE

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP6 T6.1 Deliverable D6.1</p>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public



architecture commenting on the paths incoming data or queries follow to update or query a synopsis respectively. Section 5 describes the current design of the SDE library and its status with respect to the synopses it currently supports. Some of them are tailored to the INFORE use case, while the SDE library also includes popular data summarization algorithms that are useful in broader application scenarios. Section 6 includes an experimental evaluation regarding the scalability of the SDE component, while Section 7 includes conclusive remarks and future work directions.

 European Commission Horizon 2020 European Union Funding for Research & Innovation	Project supported by the European Commission Contract no. 825070	WP6 T6.1 Deliverable D6.1	Doc.nr.: WP6 D6.1
			Rev.: 1.0
			Date: 20/12/2019
			Class.: Public



3 INFORE Requirements by the SDE Component

In this section we detail the functional and non-functional requirements that were taken into consideration during the current development of INFORE SDE component. In a nutshell, non-functional requirements define fundamental properties of a system and its components, essentially describing how a system should behave. On the other hand, functional requirements describe the actual functionality the SDE should provide to the rest of the INFORE components.

INFORE’s focus is, by design, on the real-time, online analysis of massive streaming data. In INFORE’s Life Science use case we need to analyze streams of data depicting the effect of drug combinations on cancer evolution simulations in real-time so as to stop the execution of unpromising simulations and free system resources for starting new ones. In INFORE’s Financial use case we need to monitor the behavior of or detect correlations among stocks to identify investment opportunities and support predictions on system risks, respectively. Moreover, in the Maritime use case we perform real-time anomaly detection. Although the SDE component could have been implemented in any modern Big Data platform that supports stream processing, such as Apache Spark², Flink³, Storm⁴, Kafka Streams⁵, and Akka⁶, we decided to develop the SDE based on Apache Flink. The non-functional requirements cited below describe the rationale behind this decision. Subsequently, the functional requirements, account for the requirements the rest of the INFORE architectural components pose to the design and utility of synopses, not only derived by the project’s use cases, but also from the needs of broader application scenarios that are targeted by INFORE.

3.1 Non-functional Requirements – Implementation Platform

The project prototypes and components are to be provided open source as explicitly stated in the exploitation plan of the project proposal. This applies to all the components engaged in INFORE architecture, including the SDE. Therefore, one of the key non-functional requirements concerns the corresponding licensing of the chosen implementation platform and unqualifies proprietary or commercial stream processing platform solutions.

3.1.1 Support for Lambda and Kappa Architectures

Despite the streaming nature of many application scenarios and of the project itself, it is very common for complex workflows to engage both batch processing and stream processing operations. This is also true for INFORE use cases. Our Financial use case as described in Deliverable D2.1 engages both streaming (Level 1, Level 2, Level 3) and historical data of monitored quotes. In the Maritime use case, Patterns of Life, Sentinel-1 and Sentinel-2 data are of historical nature, while derived streams of vessel metrics flow in the relevant workflows in real time. Moreover, AIS data are provided both for past and active trajectories inscribed by vessels. Workflows that engage both batch and stream processing pipelines essentially relate to a Lambda architecture as depicted in Figure 1. There, the data that are engaged in the prescribed workflows may either be stored in a (distributed) filesystem, while streaming data may be digested using a streaming framework via Kafka. Each part of the batch and stream processing is composed of one or more operations which then stream their results at a Lambda point, which may also involve a number of streaming operations, before delivering the final output streams to the respective applications.

On the other hand, in the Life Science use case, which focuses in monitoring simulations of tumor evolution under the application of different drug combinations, we may execute instances of the same simulation with different input parameters and compare the evolved simulations at runtime. The goal is to estimate which of them are related, in the sense that they produce similar results. In turn, the latter may involve the application of a series of operators quantifying cross-correlations among running simulations and classifying them based on their expected usefulness in leading cancer cells to necrosis or apoptosis. Then, we aim to keep some of the running simulations and cease the rest that are classified as non-useful so as to provision cluster resources for starting new, expectedly more

² <https://spark.apache.org/docs/latest/index.html>

³ <https://flink.apache.org/>

⁴ <https://storm.apache.org/>

⁵ <https://kafka.apache.org/documentation/streams/>

⁶ <https://akka.io/>

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP6 T6.1 Deliverable D6.1</p>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public

interesting, simulation instances. Such a scenario resembles more with a Kappa architecture as illustrated in Figure 2.

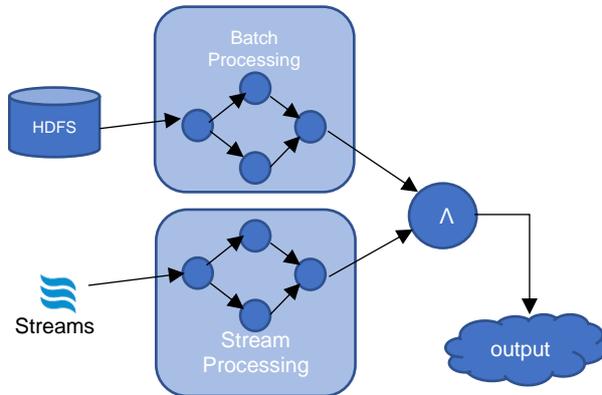


Figure 1: Lambda Architecture

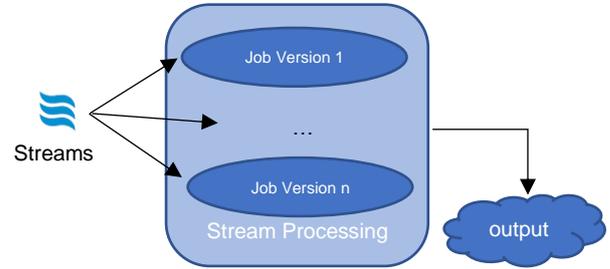


Figure 2: Kappa Architecture

INFOR workflows may involve operators that are implemented and executed in a variety of Big Data platforms. Thus, it is important for the SDE's proof-of-concept implementation prototype to build upon a platform which supports both the aforementioned architectures.

Among the popular (see Figure 3) open source Big Data platforms, Apache Spark, Flink and Akka provide support for both architectures which narrows down the choices for our proof-of-concept implementation. Spark simulates stream computing based on batch processing by adopting the concept of micro-batches in Spark Streaming⁷. Flink does the opposite, in the sense that it simulates batch processing based on streaming computation. Despite the fact that Structured Streaming⁸ has been recently added to Spark to offer similar functionality as Flink, it has neither reached the level of maturity of Flink by being adopted in a variety of real-world use cases⁹, nor it provides an equivalent support for event processing workflows as discussed below. Therefore, we henceforth exclude it from the candidate systems that will form the basis of our proof-of-concept implementation.

Package	Rank	Overall	Github	Stack Overflow	Search Results	Package	Rank	Overall	Github	Stack Overflow	Search Results
apache spark	1	20.11	8.88	6.77	4.47	apache bookkeeper	11	1.17	-0.28	-0.23	1.68
apache hadoop	2	13.33	2.03	8.20	3.11	google bigquery	12	1.12	-0.38	0.99	0.51
apache kafka	3	5.39	2.84	1.18	1.37	apache ignite	13	0.95	0.22	0.23	0.50
apache storm	4	2.86	2.18	0.24	0.44	apache zeppelin	14	0.94	1.13	-0.08	-0.11
presto	5	2.48	2.31	-0.17	0.34	metamarkets druid	15	0.91	1.70	-0.18	-0.61
stratio crossdata	6	2.03	-0.32	-0.23	2.58	concurrent cascading	16	0.91	-0.21	1.27	-0.16
apache couchdb	7	1.89	0.82	1.01	0.07	apache pig	17	0.89	-0.08	0.85	0.12
apache flink	8	1.85	1.11	0.04	0.69	apache vxquery	18	0.87	-0.37	-0.23	1.47
twitter heron	9	1.52	0.66	-0.23	1.09	apache flume	19	0.84	0.28	0.05	0.51
hazelcast	10	1.40	0.61	0.11	0.68	apache spark streaming	20	0.83	-0.23	0.65	0.41

Figure 3: Big Data platforms ranked by popularity¹⁰

⁷ <https://spark.apache.org/docs/latest/streaming-programming-guide.html>

⁸ <https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html>

⁹ <https://flink.apache.org/poweredby.html>

¹⁰ Source: <https://blog.thedataincubator.com/2018/02/ranking-popular-distributed-computing-packages-for-data-science/>

<p>Project supported by the European Commission Contract no. 825070</p>	<h2>WP6 T6.1 Deliverable D6.1</h2>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public



3.1.2 Event Processing Orientation & Ease of Use, Flexibility, Reliability

The Complex Event Forecasting (CEF) component is one of the fundamental components in INFORE’s architectural vision. In particular, the rules discovered by the online machine learning module of INFORE are deployed by the CEF component, not only to detect whether incoming data streams satisfy one or more of these rules, but also to predict a rule’s fulfillment well in advance. Therefore, the SDE should preferably be implemented on a Big Data platform with event processing orientation. The idea in this section is to elaborate on the features of each Big Data platform and finally distinguish the platform which provides native support for most of the features related to event processing. This minimizes the custom coding needed, even for fundamental stream processing facilities (such as out-of-order event arrival handling) and allows us to devote effort solely on implementing the envisioned functionality for a proof-of-concept prototype of the SDE component.

Among the three platforms that satisfy the requirement for hybrid workflows, Apache Flink provides native support for Complex Event Processing (CEP) via FlinkCEP. Another advantage of Flink is that it natively supports windowing operations on streams using event-time, i.e., all time-based operations use a timestamp attribute tagged on the streaming tuple by the producer device, typically before the tuple enters the corresponding Big Data platform. This is beyond the processing time, i.e., all time-based operations use the system clock of the machines that run the respective operations and ingestion time, i.e., the time that a tuple enters the Big Data platform. The support for event-time based processing further facilitates the native handling of out-of-order event arrivals since these are defined based on the timestamp attributed to a streaming tuple by its source.

Notably, among Spark Streaming, Flink and Akka, Flink is the only one that provides native support for event processing as outlined above, while Spark and Akka require the developer to reside to manual configurations and custom coding solutions. Despite the fact that the language of the FlinkCEP API is cumbersome, a Flink implementation of the SDE also facilitates benchmarking its utility upon cohosted with FlinkCEP against collaboratively working with CEP and CEF engines developed by project partners¹¹. Besides, it seems easier for Flink to incorporate the functionality of other popular CEP engines such as Esper¹², which is not easily doable for other Big Data platforms such as Spark [2].

At this point, we have to note that among popular streaming Big Data platforms, Apache Storm seems the most well suited for supporting a variety of CEP engines. This is because, despite Storm does not provide native support for CEP, it is more extensible to a variety of CEP engines. In particular, the standard way to incorporate CEP functionality in it, is as simple as importing CEP engine instances as libraries in Bolts of a Storm topology¹³. This has been demonstrated for CEP engines such as Esper¹⁴ and Siddhi¹⁵.

Table 1 provides a comparative analysis of Spark Streaming, Storm and Flink with respect to their “friendliness” to event processing [2]. Although Akka provides support for streaming operations in a per tuple fashion, the rest of the event-oriented features of Table 1 that refer to Spark Streaming also apply to Akka. Furthermore, for Kafka Streams the Kafka Streams CEP API¹⁶ is available to provide a subset of event processing functionality of FlinkCEP, but this time from Kafka Streams. Thus, the part of Table 1 that refers to Apache Flink also accounts for features attributed to CEP by Kafka Streams CEP.

The first row of the table refers to how CEP functionality can be attributed to each platform. In Spark Streaming, we either need to initiate CEP instances in separate cluster nodes or incorporate such functionality to executors. In a nutshell, Spark executors constitute Java Virtual Machines (JVMs) running in worker nodes of a cluster. Storm incorporates CEP functionality by importing CEP engine instances as libraries in Bolts of a Storm topology, while Flink provides native CEP support via the FlinkCEP API, as already discussed. The second row of the table, namely Event Processing Unit Size, essentially states that Spark Streaming is not a true streaming engine as it imitates stream processing using micro-batches.

¹¹ <http://cer.iit.demokritos.gr/RTEC/>

¹² <https://github.com/phil3k3/flink-esper>

¹³ <https://storm.apache.org/releases/2.1.0/Tutorial.html>

¹⁴ <https://github.com/tomdz/storm-esper>

¹⁵ <https://docs.wso2.com/display/CEP410/Creating+a+Storm+Based+Distributed+Execution+Plan>

¹⁶ <https://github.com/fhussnonnois/kafkastreams-cep>

 <p>Project supported by the European Commission Contract no. 825070</p>	<h2>WP6 T6.1</h2> <h3>Deliverable D6.1</h3>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public

Event delivery guarantees refer to how many times an event tuple will be processed in case of system failures which directly affect the reliability of CEP. Exactly-once is the strongest guarantee where each event tuple will be processed exactly once, as in the case when no failures occur. Flink provides exactly-once guarantees for FlinkCEP. At-least-once is a weaker guarantee where no event tuple will be lost but, it may be processed multiple times. Storm provides at-least-once guarantees. At-most-once means that after a system failure a tuple may be totally lost. Although Spark Streaming can provide exactly-once per se, its combination with a CEP engine to separate cluster nodes or in executors affects this guarantee [2]. An external CEP node can by default provide only the weaker at-most-once guarantee [2].

Feature	Apache Spark Streaming		Apache Storm	Apache Flink
CEP @	External Node(s)	Executor	Storm Bolt	Native CER Operator
Event Processing Unit Size	Micro-batch of Events	Micro-batch of Events	Event Tuple	Event Tuple
Event Delivery Guarantees	at-most-once	at-least-once	at-least-once	exactly-once
Windowing	Time-based	Time-based	Time, Tuple-based	Time, Tuple-based
Event Temporal Processing	processing-time	processing-time	event-time, processing-time	event-time, processing-time, ingestion-time
Out of Order Processing	No	No	Yes, using event-time	Yes, using event-time
Ease of Use	X Separate node(s) for CEP	X Manual launch of CEP engine at each Executor or via custom cluster manager	✓ Import CEP engine's functionality in Bolt's definition	✓ Native CEP API
Flexibility	XXX Micro-batch to event tuples conversion from/to the CEP engine. Requires custom handling of tuple-based windows. Only processing time-based windows	XXX	✓✓✓ Flexible, if CEP engine can be incorporated as a library in Bolt. Support for time- and tuple-based windows. Support for processing and event time-based processing	✓✓✓ Most flexible. Support for time- and tuple-based windows. Supports all event temporal processing models
Reliability	XX Due to at-most/least-once guarantees custom checks for selection/ consumption policies are needed. Unless custom cluster manager is developed. Out of order processing also requires custom code	XX	X✓ at-least-once delivery requires custom checks for selection/ consumption policies. Out of order processing supported	✓✓ Most reliable, with FlinkCEP. exactly-once guarantees. Out of order processing supported

Table 1: Event processing orientation of 3 popular Big Data platforms. The ✓ and X marks denote advantages and drawbacks, respectively, explained in the accompanying text [2].

Concerning windowed operations, contrary to Storm and Flink, Spark Streaming provides native support only for time-based windows.

The sixth row of Table 1 refers to the temporal processing model. As already discussed, Flink supports all temporal processing models. Out-of-order event arrivals may occur due to network congestion and latency on communication links delivering event tuples to the Big Data platform. The support of event-time has a direct effect on the ability of the platform to provide built-in functions for solving out-of-order event arrivals, as shown in Table 1. All three platforms define out-of-order arrivals based on event-time [2]. Due to the fact that Spark Streaming does not support event-time, it only allows the developer to use custom solutions to resolve such issues. Storm and Flink allow the specification of a slack interval parameter in built-in functions tailored to out-of-order handling. The platforms defer the computation of order-critical operators (such as windows) waiting for delayed events an amount of time equal to the slack interval. Events are buffered and re-ordered after delayed events arrive, but before an operator evaluation begins.

The second part of Table 1, i.e., the last three rows, summarizes the effect of the above features from an ease of use, flexibility and reliability viewpoint, respectively. For instance, Spark Streaming is marked as a not easy-to-use choice in the context of CEP as it requires launching external CEP nodes or CEP tasks in executors of the cluster. Moreover, it is judged as inflexible as it requires micro-batches to be converted to individual tuples before being fed

 Project supported by the European Commission Contract no. 825070	<h2>WP6 T6.1</h2> <h3>Deliverable D6.1</h3>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public



into a CEP task and vice versa. It further requires custom handling of tuple-based windows and supports only the processing-time temporal processing option. Therefore, it receives three **X** marks regarding flexibility, while Storm and Flink which (i) process each tuple individually, (ii) provide native support for tuple- and time-based windows, (iii) support event-time and processing-time temporal processing models, receive three **✓** in the corresponding fields.

The last row of Table 1 concerns reliability features for each platform based on its ability to allow exactly-once event delivery guarantees and to provide built-in support for handling out-of-order events. At-most-once and at-least-once may affect the accuracy (and, thus, the reliability) of CEP with respect to event selection or consumption policies. Please see [2] for further details on such policies. In platforms providing such guarantees one should include custom code provisions for addressing relevant reliability requirements of the application under consideration.

Among the examined platforms, Flink is the only system that combines support for hybrid workflows (Section 3.1.1) and provides numerous built-in event stream processing facilities. In that, it allows us to focus on developing the functionality of the SDE component itself, instead of first having to incorporate custom code for basic (e.g. windowing) operations.

3.1.3 Scalability Considerations

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

State-of-the-art benchmarks on streaming Big Data platforms [3] report that Flink has a better overall throughput both for aggregation and join queries when compared with Spark and Storm. This shows that Flink either being viewed as a pure streaming engine (upon compared against Storm) or as a means to support hybrid workflows (upon compared against Spark) provides increased horizontal scalability.

With respect to vertical scalability, the typical assumption of all existing Big Data platforms is that, while the data volume and the velocity of the data can be large, the number of distinct streaming sources is typically modest or small; thus, the focus is on techniques that can scale horizontally rather than vertically. For instance, the study of [3] is based only on a couple of streams. However, vertical scalability can find itself quite needed both in INFORE's use cases as well as in broader application scenarios. As a concrete example, consider the problem of tracking the highly correlated pairs of stock data streams (under various statistical measures, e.g., Pearson correlation) over N distinct, high speed data streams, where N is a very large number. While several streaming techniques (e.g., based on sketch synopses) are known for tracking the correlation of a given pair in space/time that is sublinear in the size of the streams, applying these ideas to track the full $\mathcal{O}(N^2)$ 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). Clearly, techniques that can provide vertical scaling are sorely needed for financial use case scenarios. Therefore, our aim is to use the platform with the best horizontal scalability according to recent benchmarks, as stated above, and confront vertical scalability issues. To achieve that, we implement synopsis that not only parallelizes the processing among a number of workers in Flink, but also prunes unnecessary comparisons of stock streams that are known beforehand that cannot be correlated. In particular, this non-functional requirement especially for the Financial and Life Science use cases of the project is accounted for in the way the Discrete Fourier Transform (Section 5.2.2) and Locality Sensitive Hashing summaries (Section 5.2.1) are constructed within our Flink implementation of the Synopses Data Engine.

Finally, federated scalability is taken into consideration in the SDE architectural design as we discuss in Section 4.

3.1.4 Pluggability Considerations

There are three dimensions of pluggability that are relevant to the SDE and the project. First, a synopsis provided by the SDE is essentially destined to be used as an operator of a designed workflow. Each workflow, in turn, may

 <p>Project supported by the European Commission Contract no. 825070</p>	<h2>WP6 T6.1</h2> <h2>Deliverable D6.1</h2>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public



engage streaming operators available in different Big Data platform implementations. We should thus ensure the pluggability of SDE’s summarization operators to such generic workflows, ensuring that the SDE will be able to communicate with a variety of platforms. In order to abide by this non-functional requirement, SDE’s input and output is provided via Apache Kafka¹⁷. There is a number of arguments behind the choice of Kafka.

Existing benchmarks [5] show that Kafka can serve the highest rates of data arrivals compared to other alternatives such as Apache ActiveMQ¹⁸, a popular open-source implementation of JMS, and RabbitMQ¹⁹, a message system known for its performance. Kafka is often used in real-time streaming data architectures to provide real-time analytics. Since Kafka is a fast, scalable, durable, and fault-tolerant publish-subscribe messaging system, it is used in use cases where JMS, RabbitMQ, and ActiveMQ may not even be considered due to volume and responsiveness issues.

Kafka can work with all popular Big Data platforms and other storage or messaging engines including Flume, Spark, Storm and Flink for real-time ingestion, analysis and processing of streaming data²⁰. Additionally, Kafka brokers support massive message streams for low-latency analysis in platforms supporting batch or hybrid workflows such as Hadoop or Spark, respectively. Of course, by using Kafka, Kafka Streams can be used for real-time analytics as well. Kafka’s popularity continuously grows and is nowadays a ubiquitous solution for scalable injection of Big Data²¹.

The second level of pluggability required by INFORE and ensured by Kafka involves the ability to work on top of many, potentially geographically dispersed clusters serving as a messaging cable connecting the entire federation. Finally, with respect to the third pluggability dimension, Flink Programming APIs include Java and Scala which support subtype polymorphism useful for ensuring pluggability of new synopsis in SDE’s libraries. In a nutshell, due to subtype polymorphism we allow the definition of an abstract synopsis class, which is refined by the implementation of synopsis-specific classes, one for each data summarization technique included in SDE’s libraries. Our current implementation uses Java for this purpose.

3.1.5 Other Desirable Features

Besides the above non-functional requirements, our choice of implementing the SDE on Apache Flink was motivated by the following non-functional requirements that are satisfied by it:

- Flink provides a rich set of operators including native support for iteration which is useful in supporting the Machine Learning component of the project.
- Flink provides facilities for easily connecting streams of data and requests (queries) in a way that allows access the state of currently maintained synopsis.
- With respect to memory management, Flink manages its own memory²² never breaking the JVM heap.
- Flink constitutes a Big Data platform of European origin and although lately acquired by Alibaba²³, recent H2020 projects [5] have been built on and have extended Flink’s functionality. INFORE’s SDE component thus follows this successful paradigm.

3.2 Functional Requirements

Having discussed the non-functional requirements that led us to the adoption of Flink and Kafka for our proof-of-concept SDE implementation and having reasoned about how the distributed version of certain synopses, such as those in Sections 5.2.1 and 5.2.2, should support vertical scalability, we proceed with the definition of functional requirements. The functional requirements involve the functionality the SDE needs to provide to other operators.

¹⁷ <https://kafka.apache.org/>

¹⁸ <https://activemq.apache.org/>

¹⁹ <https://www.rabbitmq.com/>

²⁰ <https://cwiki.apache.org/confluence/display/KAFKA/Ecosystem>

²¹ <https://redmonk.com/fryan/2017/05/07/the-continued-rise-of-apache-kafka/>

²² <https://ci.apache.org/projects/flink/flink-docs-stable/ops/config.html>

²³ <https://www.reuters.com/article/us-data-artisans-m-a-alibaba/alibaba-buys-german-data-analysis-start-up-idUSKCN1P30F0>

 <p>Project supported by the European Commission Contract no. 825070</p>	<h2>WP6 T6.1</h2> <h3>Deliverable D6.1</h3>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public



That is, given a workflow where a particular data summarization technique is included as a data synopsis operator, the implementation of that technique within the SDE receives input from upstream operators and may provide output streams to downstream operators of the workflow or to application interfaces. Furthermore, the SDE may need to be queried by the INFOR Optimizer regarding the available, currently maintained synopses so as to let it examine alternative workflow execution plans including equivalent approximate operators (such as count, frequency moments, self-join size estimation) in place of respective exact operators, in exchange of an expected speed up in the execution of the processed workflow.

The SDE is equipped with an internal library of currently available synopses. The current status of the library is discussed in Section 5. According to the above observations the upper level functions the SDE needs to provide are identified below.

3.2.1 Single-stream Synopsis Maintenance

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

3.2.2 Dataset Synopsis Maintenance

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

3.2.3 Multi-stream Synopsis Maintenance

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

3.2.4 Ad-hoc Querying Capabilities

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

3.2.5 Continuous Querying Capabilities

A currently maintained synopsis should be able to accept continuous queries and provide answers to downstream operators or application interfaces every time the approximated quantity is updated either due to incoming tuples that alter the maintained synopsis and/or under some windowing (time – or count–based) [2] operation.

3.2.6 Dynamic Build/Stop of Synopses

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

 <p>Project supported by the European Commission Contract no. 825070</p>	<h2>WP6 T6.1</h2> <h2>Deliverable D6.1</h2>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public



3.2.7 Providing SDE Status Report

This functional requirement expresses the need of querying the SDE component for providing a list of the currently maintained synopses and their parameters. Such functionality may be of particular utility to the INFOR optimizer, in order to acquire a list of currently running synopses and check whether it can include them in alternative execution plans, substituting equivalent, exact operators with their approximate version. Of course, when the latter is performed, there might be some accuracy loss as a tradeoff to speeding up execution time. For this reason, it is assumed that a threshold that corresponds to the accuracy loss tolerance (i.e., the maximum accuracy loss allowed) is defined in the design of the workflow.

This functional requirement is further refined to the following operations:

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

3.2.8 External Synopsis Load and Maintenance

As already mentioned at the beginning of the current section, the SDE is equipped with a library of offered, implemented data summarization techniques. The contents of the current version of this library will be discussed shortly. Besides what is included in the current version of the aforementioned library, the SDE should be able to allow the dynamic loading of synopses originating from external libraries, so as to cover all possible application scenarios where application workflows require domain specific synopses, i.e., beyond popular ones covering broad application scenarios [6] or those that are tied to the INFOR's use case. This functionality is not implemented in the first version of the SDE, but is deferred to Deliverable D6.4 on month 24 of the project.

 Project supported by the European Commission Contract no. 825070	<h2>WP6 T6.1</h2> <h2>Deliverable D6.1</h2>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public

4 SDE Architecture

In this section we detail the SDE architectural components and present their utility in serving the functional requirements identified in Section 3.2. Figure 4 provides a condensed view of the SDE architecture. Before proceeding to explaining the functionality of each component and the way they cooperate in the scope of the SDE architecture, we discuss the fundamentals of the parallelization schemes utilized within the SDE.

4.1 Employed Parallelization Scheme(s)

The parallelization scheme that is employed in the design of the SDE is key-based parallelization. That is, every data tuple that streams in the SDE architecture and is destined to be included in a maintained synopsis, does so based on the key it is assigned to it. When a synopsis is maintained for a particular stream (see functional requirements in Section 3.2.1 and Section 3.2.3) the key that is assigned to the respective update (newly arrived data tuple) is the StreamID of that particular stream for which the synopsis is maintained. In this case, within the distributed computation framework of Flink, that stream is processed by 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 hosting the SDE. On the other hand, when a synopsis involves a dataset in its entirety (see respective functional requirement in Section 3.2.2) the desired degree of parallelism is included as a parameter in the respective request to build/start maintaining the synopsis (see function requirement in Section 3.2.6). In the latter case, one dataset is partitioned to the available workers in a round-robin fashion and the respective keys are created by the SDE, as we describe in detail later on, each of which corresponds to a particular worker.

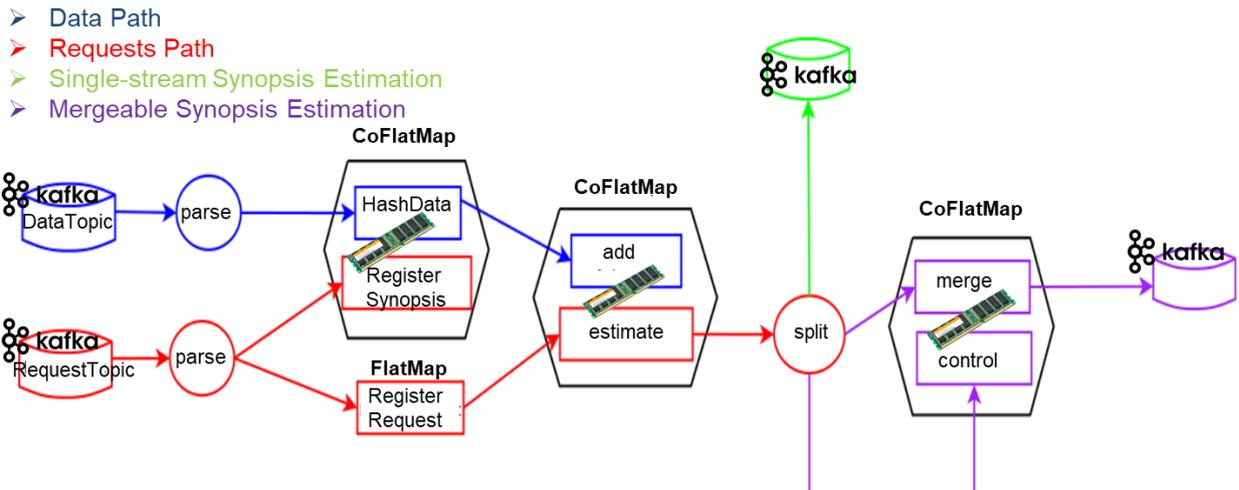


Figure 4: SDE Architecture -- Condensed View

There are two important remarks that we need to make at this point. The first one links the key assignment in the case of Dataset Synopsis Maintenance (Section 3.2.2) with the nature of the maintained synopsis. In order to distribute the load among the available workers in the way described above, the synopsis itself needs to possess the mergeability property [7]. Mergeability, refers to the ability of building synopsis on parts of the data and then having a way to merge the partial synopses into one synopsis, which will be equivalent to the synopsis that would have been built in case the synopsis was maintained centrally (instead of distributively). For instance, in case FM sketches or Bloom Filters [8] (bitmaps) are built on different data partitions at separate worker nodes, they can be merged into one FM sketch or Bloom Filter via simple logical disjunction (OR) or conjunction (AND) operations. In case a maintained synopsis does not possess the mergeability property, the corresponding synopsis maintenance request should be parameterized with a unitary parallelization degree. The second remark we need to make here is that in the current design of the SDE, this is supposed to run on a number of worker nodes of a cluster, which poses an upper limit on the possible parallelization degree.

 Project supported by the European Commission Contract no. 825070	<h2>WP6 T6.1</h2> <h3>Deliverable D6.1</h3>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public



4.2 Data and Query Ingestion

Having clarified the above, we proceed with describing the flow of data and of the queries (requests) in the SDE architecture shown in Figure 4. Data and request streams arrive at a particular Kafka topic each. In the case of the DataTopic of Figure 4, a parser component is used in order to extract the value of the field(s) on which a currently running synopsis is maintained. The respective parser of the RequestTopic topic reads the request and processes it. When an incoming request involves the maintenance of a new synopsis, the parser component extracts information about synopsis parameters (see for instance Table 2) and nature, i.e, whether it is on a single stream, on a dataset, or involves a multi-stream synopsis maintenance request. In case the request is an ad-hoc query (see functional requirement in Section 3.2.4) the parser component extracts the corresponding synopsis identifier(s).

4.3 Requesting New Synopsis Maintenance

When a request is issued for maintaining a new synopsis, it initially follows the red-colored paths of the SDE architecture. That is, the corresponding parser sends the request to a FlatMap operator (termed RegisterRequest at the bottom of Figure 4) and to another FlatMap operator (RegisterSynopsis) which is part of a CoFlatMap one. In a nutshell, a FlatMap operator takes one tuple and produces zero, one, or more tuples, while a CoFlatMap operator hosts two FlatMaps that can share the state of common variables (therefore the linking icon in the figure) among streams that have previously been connected (using a Connect operator in Flink²⁴). RegisterRequest and RegisterSynopsis produce the keys as analyzed in Section 4.1 for the maintained synopsis, but provide different functionality. The RegisterRequest operator uses these keys in order to later decide which worker(s) an ad-hoc query, which also follows the red-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 should be directed. Such an update follows the blue-colored path in Figure 4. The possible parallelization degree of the corresponding RegisterSynopsis and RegisterRequest operators, beyond being affected by the number of available worker nodes, is restricted by the number of maintained synopses.

4.4 Updating the Synopsis

When a data tuple destined to update one or more synopses is injected via the DataTopic of Kafka it follows the blue-colored path of the SDE architecture. The tuple is directed to the HashData FlatMap of the corresponding CoFlatMap where the keys (StreamID for single stream synopsis and/or WorkerID for Dataset Synopsis) are looked up based on what RegisterSynopsis has created. Following the blue-colored path, the tuple is directed to an add FlatMap operator which is part of another CoFlatMap. The add operator updates the maintained synopsis as prescribed by the algorithm of the corresponding technique. For instance, in case an FM sketch is maintained, the add operation hashes the incoming tuple to a position of the maintained bitmap and turns the corresponding bit to 1 if it is not already set. Notice, that the blue-colored path in Figure 4 remains totally detached from the red-colored path. This depicts a design choice we follow for facilitating ad-hoc querying capabilities. That is, since the data updates on several maintained synopses may arrive at an extremely high rate, typically a lot higher than the rate at which ad-hoc queries are issued, in case of the two paths were crossing at some point of the architecture, back-pressure on the blue-colored path would also stall the execution of ad-hoc queries. Having kept the two paths independent, data updates and ad-hoc queries are inserted in different processing queues and thus, even when the queue of the data updates grows, ad-hoc queries can be answered in a timely manner, based on the current status of the maintained synopses.

4.5 Ad-hoc Query Answering

An ad-hoc query arrives via the RequestTopic of Kafka and is directed to the RegisterRequest operator. The operator, which produces the keys in the same way as RegisterSynopsis does, looks up the key(s) of the queried synopsis and directs the corresponding request to the estimate FlatMap operator of the corresponding CoFlatMap. The estimate operator reads via the shared state the current status of the maintained synopsis and extracts the estimation of the corresponding quantity the synopsis is destined to provide. For instance, upon performing an ad-hoc query on an FM sketch, the estimate operator reads the maintained bitmap, finds the lowest position of the unset bit and provides a distinct count estimation using the index of that position and a $\phi=0,77$ factor. Table 2 provides a summary of the estimated quantities each of the currently supported synopsis can provide.

²⁴ <https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/>

 <p>Project supported by the European Commission Contract no. 825070</p>	<p>WP6 T6.1 Deliverable D6.1</p>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public

4.6 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 synopsis is updated by an add operation. Either way, in this particular occasion the data and request paths mandatorily cross at the second level of CoFlatMap and estimate needs to be invoked by add.

Both in ad-hoc and continuous querying, the result of the estimate is directed to a Split operator. The split operator directs the output of single stream synopses to downstream operators of the executed workflow via Kafka (green-colored path in Figure 4) or, for mergeable summaries, the output of each worker is directed to a third level CoFlatMap following the deep purple-colored path. There, the merge operator merges the partial results of the various workers and produces the final estimation which is streamed to downstream operators, again via Kafka. In order to direct all partial estimates to the same worker, a corresponding identifier for the issued request (for ad-hoc queries) or an identifier for the maintained synopsis (for continuous queries) is used as a key.

Finally, the second FlatMap termed as Control in the aforementioned CoFlatMap is included as a provision for future extensions of the SDE, over multi-cluster settings, where the synopsis that needs to be merged originates from multiple, potentially geographically dispersed nodes, each executing a separate instance of the SDE component.

Synopsis	Estimate/Output	Mostly Used for	Parameters
CountMin	Count	Frequent Itemsets	ϵ , δ , seed
BloomFilter	Set Membership	Membership	Number Of Elements, False Positive rate
FMSketch	Distinct Count	Distinct Count	Bitmap size, ϵ , δ
AMS	L_2 -Norm, Inner Product, Count	L_2 -Norm	Depth, Buckets
DFT	Fourier Coefficients/BucketID	Correlation	Window Size, Slide Size, Interval, Number of Coefficients
LSH	BucketID, Bitmap	Correlation	Similarity Function, Bitmap Size, Tumble Size, Number of Buckets
LossyCounting	Count, Frequent Items	Frequent Itemsets	ϵ
StickySampling	Frequent Items, isFrequent, Count	Frequent Itemsets	s , ϵ , δ
GKQuantiles	Quantile	Quantiles	ϵ
HyperLogLog	Distinct Count	Cardinality	Relative Error
ChainSampler	Sample of the data	Sampling	Sample Size, Window Size
STSampler	Sample of positions	Sampling	Angle, Direction, Distance, Time Interval Thresholds

Table 2: Supported synopses in Data Synopsis Generator V1. ϵ (epsilon) stands for approximation error bound, while δ (delta) stands for the probability of failing to achieve Epsilon accuracy

 <p>Project supported by the European Commission Contract no. 825070</p>	<h2>WP6 T6.1</h2> <h3>Deliverable D6.1</h3>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public

5 The Synopses Library

5.1 Structure of the Synopses Library

The internal structure of the synopses library is illustrated in Figure 5 which also provides only a partial view of the supported synopses for readability purposes. Table 2 provides a full list of currently supported synopses, their utility in terms of approximation quantities and their parameters. The development of the SDE library exploits subtype polymorphism in order to ensure the desired level of pluggability for new synopses definitions (see Section 3.1.4).

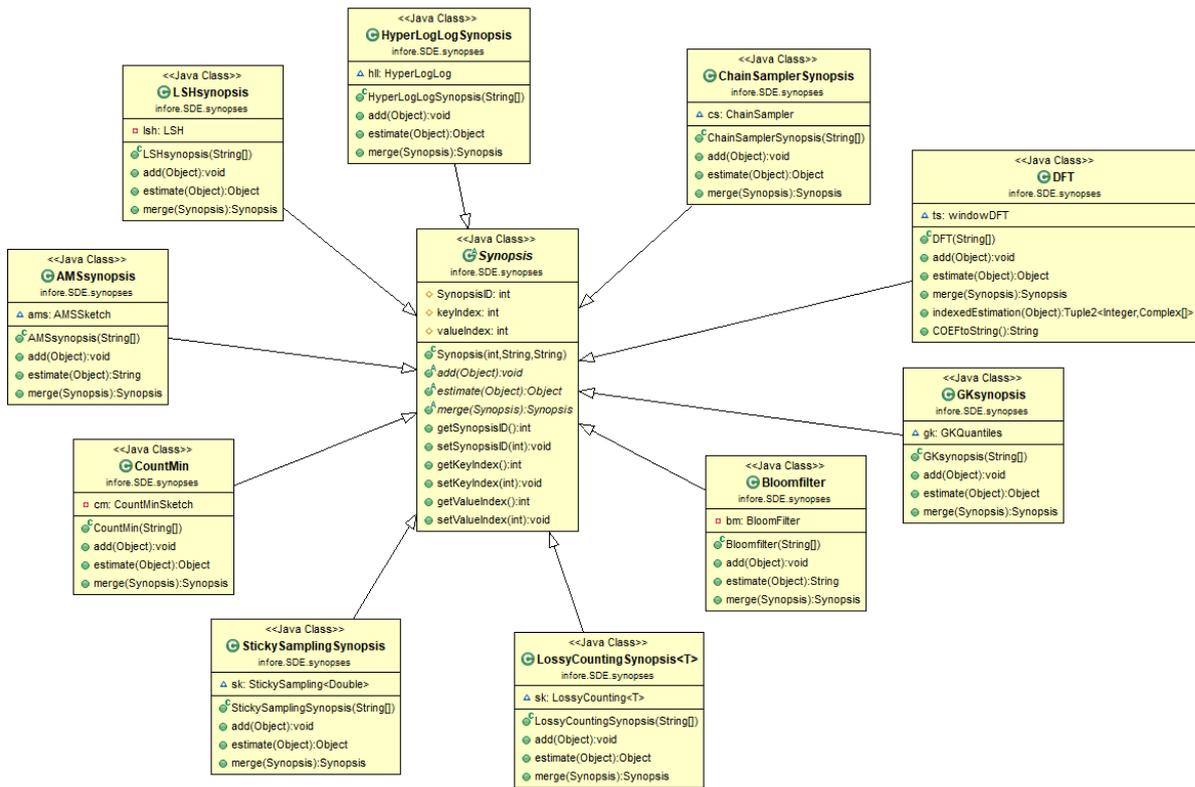


Figure 5: Structure of the Synopses Library (partial view)

As shown in Figure 5, there is a higher level class called Synopsis with attributes related to a unique identifier, a String that includes information about the index of the key field in an incoming data tuple (for single stream synopsis as defined per Section 3.2.1 or 3.2.3) as well as the respective index of the value field, i.e., the field which the summary is built on. The constructor of the class receives another String including parameters related to the identifier of a particular type of synopsis as well as synopsis specific parameters as included in Table 2. 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 5, that extends Synopsis and overrides the add, estimate and merge methods with the algorithmic details of that particular technique.

5.2 Supported Synopses Tied to INFOR Use Cases

In this section we describe the operation of a triplet of synopsis that are to support INFOR use cases. The Random Hyperplane Projection (RHP) Locality Sensitive Hashing (LSH) and the Discrete Fourier Transform (DFT) schemes we incorporate in the SDE library and describe here are of particular utility in both the Financial and the Life Science use cases. Moreover, the Spatiotemporal Sampling (STSampling) technique that we introduce serves the needs of the Maritime use case and was developed by MarineTraffic.

<p>Project supported by the European Commission Contract no. 825070</p>	<p>WP6 T6.1 Deliverable D6.1</p>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public



In the financial use case, in order to predict systemic risks or provide suggestions for investment opportunities we need to identify highly correlated stocks under a given correlation threshold. In the Life Science use case, we need to judge the similarity of the outcomes of currently running tumor evolution simulations and decide, in an online fashion, which of them are highly similar (again under a threshold) and useful (e.g. due to the fact that many cells are led to necrosis and apoptosis while few proliferate), versus those that are highly similar but non-useful. The latter category of simulations can be relatively safely stopped in order to free cluster resources that can then be devoted to new simulations where different drug combinations within different time intervals are applied. Please see Deliverable D1.1 and Deliverable D2.1 for further details.

The contribution of the RHP and DFT synopses in the Life Science and Financial use case is two-fold. First, they both perform dimensionality reduction so that all the corresponding observations that are included in a currently examined window can be summarized using a bitmap of substantially lower dimensionality or a few (up to 8) Fourier Coefficients, respectively. This harnesses memory utilization and speeds up individual, pairwise similarity test for stocks and simulation time series, respectively. Additionally, we show that using the outcomes of these techniques, i.e., the resulted bitmaps or coefficients, enables us to divide stocks or simulations into buckets that are processed by different processing units (thus parallelizing similarity checks under the given threshold) and restrict the number of required similarity comparisons to one or few buckets. The latter holds because highly uncorrelated stocks or simulation streams are assigned to different buckets using locality (i.e., similarity) aware hashing.

In the Maritime use case, thousands of vessel trajectories are simultaneously monitored, and we wish to extract simple events such as turn, acceleration, route deviation in order to deduce complex events such as piracy or illegal fishing activities. Keeping the whole trajectory of reported AIS positions of every vessel that is not associated with an event of interest exacerbates memory utilization. At a second level of analysis, should we want to detect events linked with more than one trajectory, like proximity events, comparing the whole sets of reported AIS positions of each engaged vessel increases the computational complexity of spatiotemporal joins. Deliverable D3.1 provides further details on the MSA use case and requirements. The STSampler synopsis that we describe below addresses the aforementioned challenges by reducing memory consumption per vessel trajectory as well as reducing the computational complexity of proximity or other events linked with more than one trajectory.

5.2.1 Random Hyperplane Projection (RHP) Locality Sensitive Hashing (LSH)

The RHP LSH scheme as utilized in the SDE [9][10] operates over tumblers, i.e., disjoint windows of a data stream of $|W|$ size each. Given a tumbler w_i of observations for a stream i , RHP produces a bitmap of d dimensionality with $d \ll \text{bit}(w_i)$, where $\text{bit}(w_i)$, is the size in bits of the respective values in w_i . For instance, for a window composed of doubles, the size in bits of each of them may be 64.

In order to produce such a bitmap, the RHP scheme uses a matrix R . Each column of R is a spherically symmetric random vector r_c , $c \in \{1, \dots, d\}$ of unit length and dimensionality $|W|$. R is composed of d columns. Recall that d is the desired bitmap size and it directly relates to the accuracy of the similarity tests that will be performed using the resulted projections. Please see [10] for further details.

For each r_c , $c \in \{1, \dots, d\}$ we perform $w_i \cdot r_c$ and if the result of the multiplication is positive the corresponding position of the bitmap X_i is set, or zero otherwise. According to [9][10], by producing the bitmaps in the aforementioned way, should we use them for similarity estimation, on expectation we get:

$$\frac{D_h(X_i, X_j)}{d} = \frac{\theta(w_i, w_j)}{\pi} \Leftrightarrow \theta(w_i, w_j) = \frac{D_h(X_i, X_j)}{d} \pi \stackrel{0 \leq \theta(w_i, w_j) \leq \pi}{\Leftrightarrow} \cos(\theta(w_i, w_j)) = \cos\left(\frac{D_h(X_i, X_j)}{d} \pi\right)$$

where $D_h(X_i, X_j) = \sum_{k=1}^d |X_{ik} - X_{jk}|$ is the Hamming distance of the involved bitmaps. The above equation describes that, viewing the tumbler of each stream as $|W|$ -dimensional vector, on expectation the angle among such vectors originating from a pair of streams is analogous to the Hamming distance of their RHP bitmaps. Furthermore, it provides a way to compute the Cosine Similarity of the initial vectors using the bitmaps. Therefore, a first supported metric for similarity estimation is the Cosine Similarity.

As shown in [9][10], the Cosine Similarity among $w_i^* = w_i - E(w_i)$, $w_j^* = w_j - E(w_j)$, (where $E()$ stands for the expected/average value of the vectors) is directly related to the Pearson's Correlation Coefficient (Corr), i.e.,

 <p>Project supported by the European Commission Contract no. 825070</p>	<h2>WP6 T6.1</h2> <h3>Deliverable D6.1</h3>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public

$Corr(w_i, w_j) = Corr(w_i^*, w_j^*) = \cos(w_i^*, w_j^*)$. Thus, the Correlation Coefficient is another supported similarity metric, while similarity based on Euclidean Distance or Tanimoto Coefficient can also be deduced using the produced bitmaps [10].

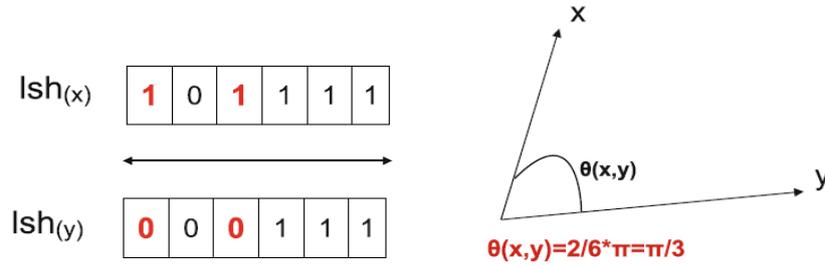


Figure 6: Estimating the angle among vectors x,y using the Hamming distance of their associated RHP footprints [11].

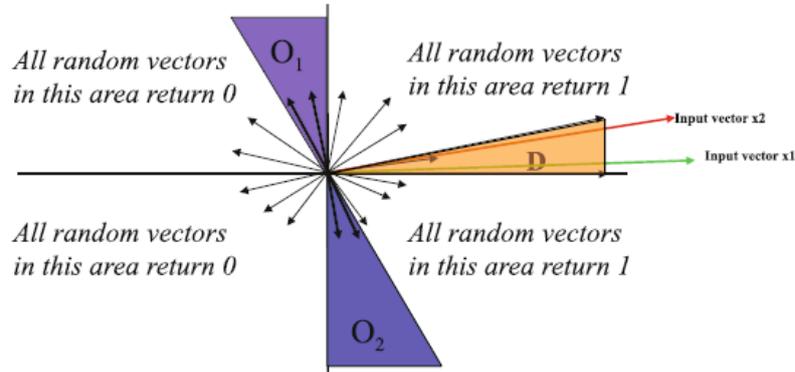


Figure 7: Operation of RHP using bitmaps of length 1. Only random vectors in the shaded areas O1 and O2 (equivalent to D, but around the hyperplane inscribed by the single random vector) can help distinguish between different data items [11].

So, initially, the idea in order to reduce memory utilization and simplify the nature of the similarity test to simple Hamming distance estimation is to produce the bitmap of each stream. Using these bitmaps, the algorithm computes the Hamming distance of each pair of stream windows and compares the resulted Hamming distance against a Hamming Threshold. To do so, the application needs to provide a similarity threshold T which will be progressively interpreted to an angle T_{angle} and then a Hamming Distance threshold T_h , using the above equation, i.e.

$$T = \cos\left(\frac{T_h}{d}\pi\right) \quad \Leftrightarrow \quad T_{angle} = \arccos(T) = \frac{T_h}{d}\pi \Leftrightarrow T_h = \frac{T_{angle}}{\pi}d$$

If $D_h(X_i, X_j) > T_h$ the corresponding streams are classified as dissimilar or considered similar otherwise. Let us now see how we prune pairwise stream comparisons for similarity and simultaneously parallelize the processing among the available processing units, thus enhancing vertical scalability. The whole idea involves hashing the produced bitmaps to buckets using their Hamming weight. The Hamming weight of a bitmap is given by: $W_h(X_i) = \sum_{k=1}^d |X_{ik}|$, i.e., it is simply the count of set bits.

 Project supported by the European Commission Contract no. 825070	<h2>WP6 T6.1</h2> <h3>Deliverable D6.1</h3>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public

An important observation is that $D_h(X_i, X_j) \geq |W_h(X_i) - W_h(X_j)|$ and thus if $T_h \geq |W_h(X_i) - W_h(X_j)|$ the bitmaps X_i, X_j are surely dissimilar and can thus be hashed to different buckets and avoid to be compared for similarity. If we wish a maximum parallelization degree of $B < \frac{d}{T_h}$ buckets, then we hash the bitmaps to B buckets as follows:

- Each bitmap is hashed to the $\left\lfloor \frac{W_h(X_i)}{\frac{d}{B}} \right\rfloor$ -th bucket
- For checking the similarity of bitmaps whose Hamming weight is near the edges of each bucket and thus they may have $T_h \geq |W_h(X_i) - W_h(X_j)|$, under the restriction of $B < \frac{d}{T_h}$, we also hash the bitmap to the $\left\lfloor \frac{\max\{W_h(X_i) - T_h, 0\}}{\frac{d}{B}} \right\rfloor$ -th bucket, provided it is different than the $\left\lfloor \frac{W_h(X_i)}{\frac{d}{B}} \right\rfloor$ -th one.
- To avoid duplicate similarity estimations, for bitmaps initially hashed to the same bucket, the similarity test between them is performed only in that bucket node.

Note that the principal role of the SDE component is to produce the corresponding bitmaps and hash them to buckets. Therefore, the output of the corresponding synopsis in Table 2 includes these two results of RHP, the bitmap and the bucket it is hashed to. The actual similarity tests are described here for completeness of presentation. In practice, similarity tests may be performed by some downstream operator in the designed workflow, such as an online clustering algorithm of the Machine Learning component of INFORE.

5.2.2 Discrete Fourier Transform (DFT)

We utilize DFTs as an alternative to the LSH algorithm of Section 5.2.1, in order to approximate the correlations between pairs of time series (e.g., from the Financial or Life Science use case). Our DFT-based correlation estimation implementation is mostly based on StatStream [12]. We note beforehand that there is a direct relation between Pearson's correlation coefficient among time series x, y and the Euclidean distance of their corresponding normalized version (we use primes to distinguish DFT coefficients of normalized time series from the ones of the unnormalized version). In particular, $Corr(x, y) = 1 - \frac{1}{2}d^2(X', Y')$ [10][12], where $d(\cdot)$ is the Euclidean distance.

The Discrete Fourier Transform transforms a sequence of N 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:

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

Compression is achieved by restricting F in the above formula to few coefficients. In our implementation we set $F=0, \dots, 7$, i.e., we use up to 8 coefficients for comparing time series, after performing an exploratory analysis on the time series present in INFORE-related datasets and because of the fact that the majority of the Fourier signal energy is concentrated on the first few coefficients [14].

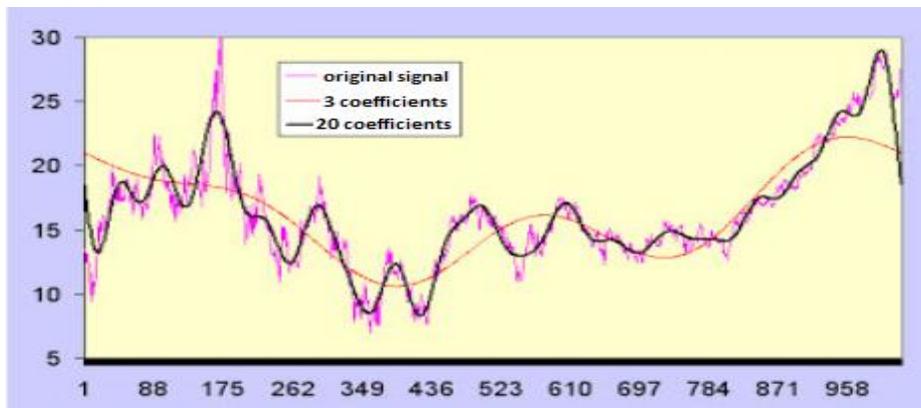


Figure 8: Approximating time series with DFT using 3 and 20 Coefficients [13].

 <p>Project supported by the European Commission Contract no. 825070</p>	<h2>WP6 T6.1</h2> <h3>Deliverable D6.1</h3>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public

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

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

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

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

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

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

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

Based on the above observation, [12] claims that the range of each DFT coefficient is between $-\frac{\sqrt{2}}{2}$ and $\frac{\sqrt{2}}{2}$. Therefore, the DFT feature space is a cube of diameter $\sqrt{2}$. Based on this, we use a number of DFT coefficients to define a grid structured, composed of cells/buckets for hashing groups of time series to each of them. Each cell is the grid is of diameter ε and there are in total $2 \left\lceil \frac{\sqrt{2}}{2\varepsilon} \right\rceil^{\#used_coefficients}$ buckets.

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

Again, note that the principal role of the SDE component is to produce the corresponding DFT coefficients and hash them to buckets. Therefore, the output of the corresponding synopsis in Table 2 includes the resulted coefficients and the bucket identifier. The actual similarity tests (in each bucket) may be performed by some downstream operator in the designed workflow.

 <p>Project supported by the European Commission Contract no. 825070</p>	<h2>WP6 T6.1</h2> <h3>Deliverable D6.1</h3>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public

5.2.3 STSampler Synopsis

The idea behind the STSampler synopsis is that when no important differentiation of a vessel's currently inscribed trajectory is observed, we may avoid keeping the respective positions in memory and instead sample only positions within the trajectory that are considered important, since they cannot be deduced or approximated sufficiently well by interpolating other sampled positions. The STSampling scheme that we incorporate in the SDE library resembles the concept of threshold-guided sampling discussed in [15], but approaches the sampling process in a more simplistic, yet tailored to the MSA application needs, way.

The sampling process is executed in a per stream fashion, i.e., for the currently monitored trajectory of each vessel separately. The core concept of the sampling process is that if the velocity and the direction of the movement of the vessel does not change significantly, the corresponding AIS message is not sampled.

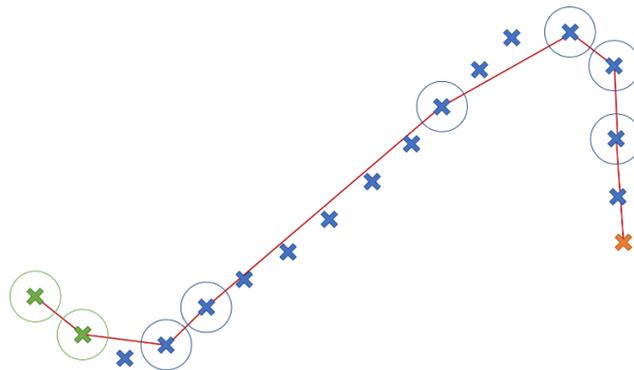


Figure 9: Example illustration of STSampler.

The last two reported trajectory positions are cached. When an AIS message holding information about the current status of the vessel streams in, we compute the change in the velocity between the lastly cached and the new AIS report, i.e. $\Delta_{vel} = |vel(prev) - vel(now)|$ and compare this value against a velocity threshold T_{vel} . Using the previously cached points we compute the vector describing the lastly reported direction of the vessel $dir(prev)$, while using the last cached and the newly reported positions we compute $dir(now)$. Then we compare $\Delta_{dir} = |dir(prev) - dir(now)|$ against a direction threshold T_{dir} . If at least one of these deltas does not exceed the corresponding threshold, the newly received AIS message is not included in the sample. This holds provided that a couple of additional spatiotemporal constraints are satisfied: (a) the time difference between the newly received AIS message and the last one that was included in the sample does not exceed a given time interval threshold T_{diff} and (b) the distance among the most recently sampled and the current position of the vessel does not surpass a distance threshold T_{dist} .

In Figure 9 ship positions (depicted using a cross) within a circle are sampled because they represent a change in the movement. The first two positions received (depicted in green) are always sampled because they are required for initialization. The last position received (depicted in orange) is considered as sampled until a new position is received.

5.3 Supported Synopsis for Broader Application Scenarios

In this section we discuss the functionality of synopsis techniques that are incorporated in the current version of SDE's library and utilized for providing approximate answers with respect to quantities that are frequently used in a variety of broader application scenarios. Such quantities involve count, distinct count, frequency moment estimation, as set membership and samples (which in turn provide estimations of average, variance etc) of data streams.

5.3.1 Lossy Counting

The Lossy Counting algorithm maintains a data structure, which is a set of entries of the form $\langle element, frequency, \epsilon \rangle$, where *element* is a data element, *frequency* is an integer representing the estimated frequency of the element and ϵ tunes the allowed maximum possible error in frequency estimation. According to the Lossy Counting rationale

	Project supported by the European Commission Contract no. 825070	<h2>WP6 T6.1</h2> <h3>Deliverable D6.1</h3>	Doc.nr.:	WP6 D6.1
			Rev.:	1.0
			Date:	20/12/2019
			Class.:	Public

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

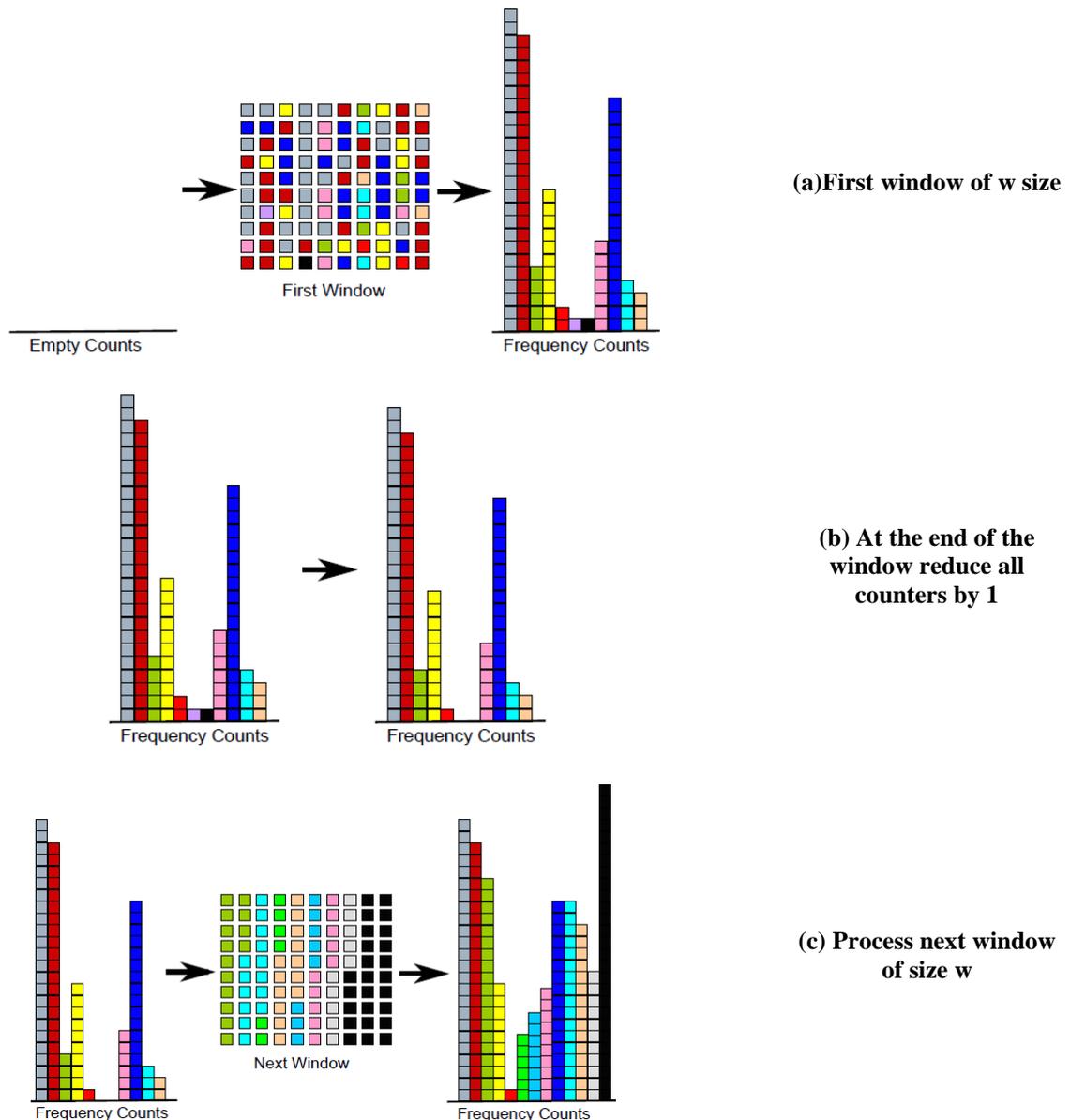


Figure 10: Lossy Counting over a window of colours²⁵

²⁵ <https://pdfs.semanticscholar.org/e7b1/11b6416107f81ebf61724eb0a2f5c7a48438.pdf>

<p>Project supported by the European Commission Contract no. 825070</p>	<h2>WP6 T6.1</h2> <h3>Deliverable D6.1</h3>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public

5.3.2 Sticky Sampling

Sticky Sampling [17] shares similarities with Lossy Counting, but differs in that (i) the size of buckets/windows is not steady and (ii) the count of an element is maintained with a certain sampling probability. This introduces a probability δ for the estimated frequencies to be less than the true frequencies by more than ϵN . The input of the algorithm is composed of a support parameter s and the desired (ϵ, δ) guarantees. The algorithm works as follows: the incoming stream is split into windows. The first window is of $w = t = \frac{1}{\epsilon} \log(\frac{1}{s\delta})$ size and this size is doubled in each subsequent one, i.e., $w = 2t$, $w = 4t$ and so on. For each window, Sticky Sampling goes through elements and if a counter for an element exists, it increases it by one each time the element is observed. If a counter for an element does not exist, one is created and initialized to 1 with probability $\frac{t}{w}$. At the end of each window, for each element's counter a coin is tossed. If the result of the coin flip is 0 the counter is reduced by one. Otherwise, the process continues by flipping a coin for the counter of the next element. If a counter of an element becomes zero, the element is dropped. The guarantees mentioned for Lossy Counting also hold for Sticky Sampling, except for a failure probability δ as described in point (ii) above.

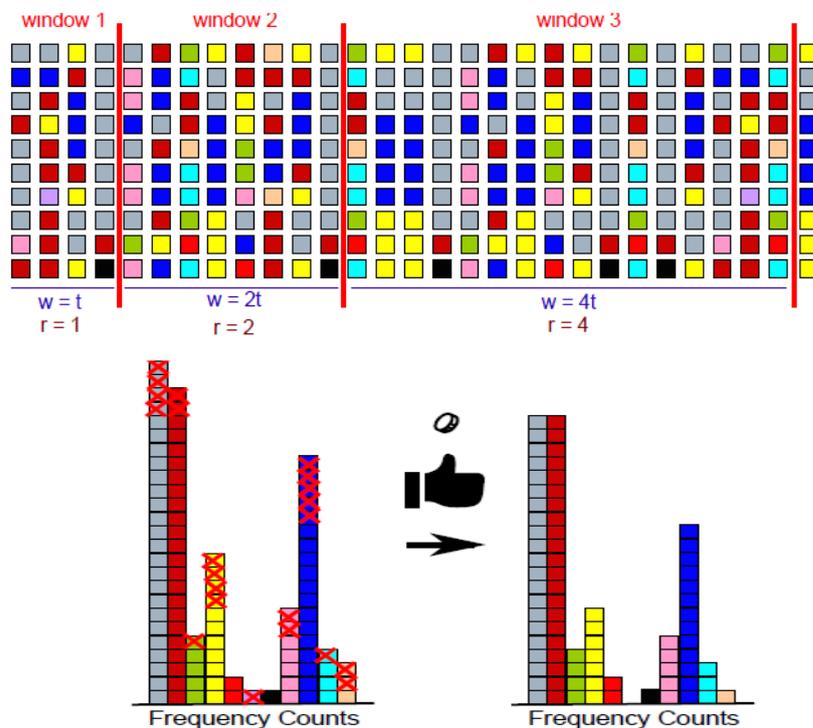


Figure 11: Sticky Sampling over a window of colors²⁶.

5.3.3 CountMin Sketch

A Count-Min Sketch [16] is a two dimensional array of $w \times d$ dimensionality used to estimate frequencies of elements of a stream using limited amount of memory. For given accuracy ϵ and error probability δ , $w = \frac{e}{\epsilon}$ and $d = \log \frac{1}{\delta} \cdot d$ random, pairwise independent hash functions are chosen for hashing each element to a column in the sketch. When an element streams in, it goes through the d hash functions so that one counter in each row is incremented. The estimated frequency for any item is the minimum of the values of its associated counters. This provides an estimation within ϵN with probability at least $1 - \delta$. For each new element, if its frequency is greater than a required threshold, it is added to a heap. At the end, all elements whose estimated count is still above the threshold are

 Project supported by the European Commission Contract no. 825070	<h2>WP6 T6.1</h2> <h2>Deliverable D6.1</h2>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public

output. In the implemented version of CountMin sketch no such threshold is employed, thus every counter of a queried element is outputted.

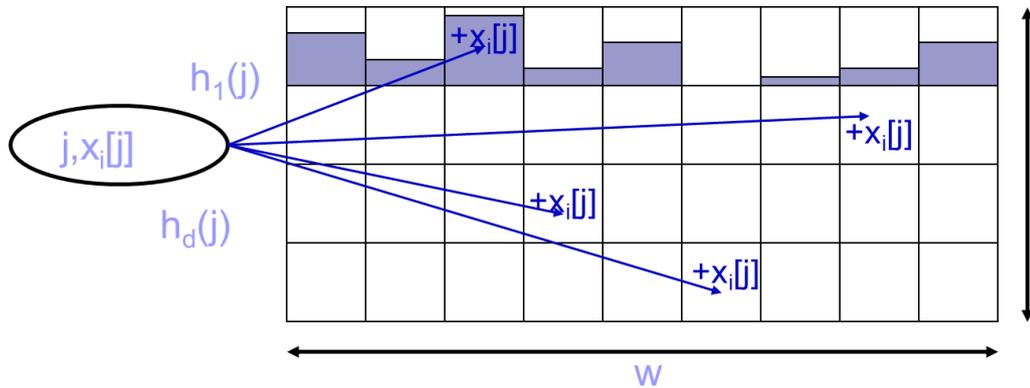


Figure 12: CM Sketch Structure. Each entry in vector x is mapped to one bucket per row. Two sketches can be merged via entry-wise summation. $x_i[j]$ is estimated by taking $\min_k \text{sketch}[k, h_k(j)]$ above [18].

5.3.4 FM Sketch

The FM sketch [20] is a bitmap B used to estimate the number of distinct elements in a stream using a limited amount of memory. The bitmap is of length $|B| = \log U$, where U is an upper bound on the number of distinct elements and all bits are initially unset. Typically, 32 or 64 bits will suffice. For one instance of the bitmap, a hash function h maps every element i to a pseudo-random integer $h(i)$ corresponding to a position in the $|B|$ -bit sketch that will be set. The values are mapped by h according to a geometric distribution, that is, the probability that i will be mapped to a position v is $P[h(i) = v] = 2^{-v}$ for $v \geq 1$. We pick $h'(i) = a \cdot i + b \text{ mod } U$ where a, b are randomly chosen and $h(h'(i))$ gives the number of trailing zeros of the binary representation of the result of $h'(i)$.

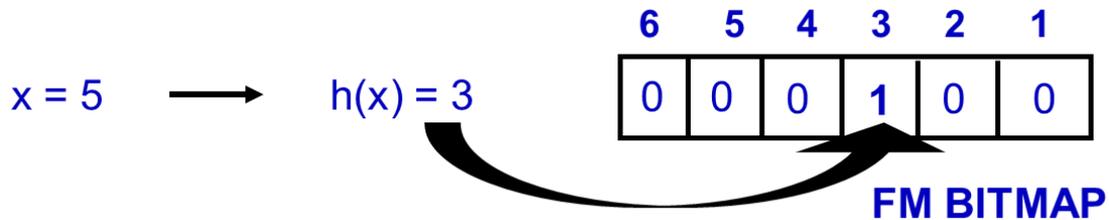


Figure 13: FM Sketch – Exemplary Update Process [18]

A distinct count estimator is built by taking k such bitmaps equipped with different hash functions and find the mean position of the leftmost zero across these repetitions as r . The overall distinct element count is estimated with 1.29×2^r . The variance is proportional to $\frac{1}{\sqrt{k}}$. By taking $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ sketches, the resulting distinct count estimation \hat{n} is within ϵn of the true distinct count n with probability at least $1 - \delta$. To avoid the $O(k)$ cost for each update that the above estimator entails, an alternative is to maintain k FM sketches out of which only one is updated per incoming stream element. FM sketches are mergeable summaries, i.e., applying a bitwise OR operator among FM sketches produces the FM sketch of the union of the observed elements.

5.3.5 Bloom Filters

A Bloom filter [21] is a space-efficient representation of a stream of n elements from a universe U , mainly used to deduce whether a certain element has been observed (set membership). A Bloom filter is a bitmap of m bits equipped with a family of k independent hash functions which hash stream elements to positions of the bitmap. The whole bitmap is initially unset. An element is inserted into the Bloom filter by setting all positions of the bitmap

 Project supported by the European Commission Contract no. 825070	WP6 T6.1 Deliverable D6.1	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public

where the k hash functions point to, to 1. At query time, an element is assumed to be contained in the original set of observations if all hashed positions of the Bloom filter are equal to 1. If at least one of these positions is set to 0, then we conclude that the element is not present.

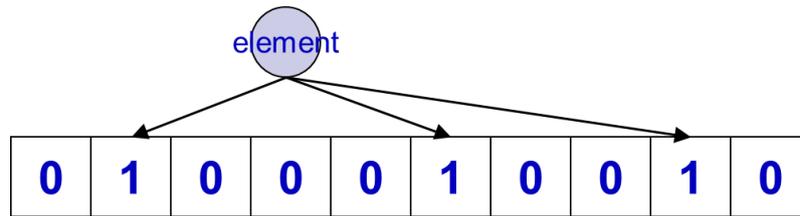


Figure 14: Bloom filter operation. $k=3$ hash functions map items to bit vector k times, set all k entries to 1 to indicate item is present [18].

Bloom filters exhibit a small probability of false positives; due to hash collisions, it is possible that all bits representing a certain element have been set to 1 by the insertion of other elements. The probability for a false positive is $\sim (1 - e^{-kn/m})^k$. For given n and Bloom filter length, the false positive probability can be minimized by optimizing the ratio between true bits and Bloom filter length. We denote this ratio as Bloom filter density. The false positive probability is minimized when this density is 0.5. This is the case when the number of hash functions is set to $k \approx m/n \ln(2)$. Equivalently, for given n , false positive probability and $k \approx m/n \ln(2)$ we can compute the required size of the Bloom filter.

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

5.3.6 HyperLogLog Sketch

The HyperLogLog algorithm constitutes the evolution of FM sketches [20] and the LogLog algorithm [23]. It is a simple, elegant algorithm that enables to extract distinct counts using limited memory and a simple error approximation formula. In the common implementation of HyperLogLog, each incoming element is hashed to a 64-bit bitmap. The hash function is designed so that the hashed values closely resemble a uniform model of randomness, i.e., bits of hashed values are assumed to be independent and to have $\frac{1}{2}$ probability of occurring each.

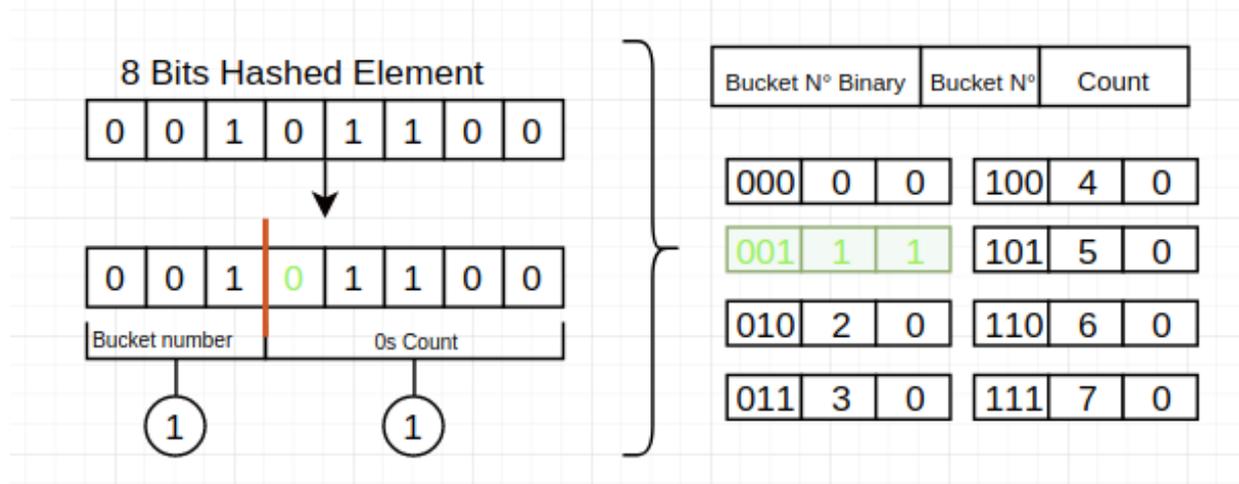


Figure 15: HyperLogLog maintenance example

The first m bits of the bitmap are used for bucketizing the element and we have an array M of 2^m buckets (also called registers). The rest $64-m$ bits are used so as to count the number of leading zeros and in each bucket we store the maximum such number of leading zeros to that particular bucket. To extract a distinct count estimation, we average

 <small>Horizon 2020 European Union Funding for Research & Innovation</small>	Project supported by the European Commission Contract no. 825070	<h2 style="margin: 0;">WP6 T6.1</h2> <h3 style="margin: 0;">Deliverable D6.1</h3>	Doc.nr.:	WP6 D6.1
			Rev.:	1.0
			Date:	20/12/2019
			Class.:	Public

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 the above procedure and equivalent number of buckets maintained independently.

5.3.7 AMS Sketch

The key idea in AMS sketch [24] is to represent a streaming (frequency) vector v using a much smaller sketch vector $sk(v)$ that is updated with the streaming tuples and provide probabilistic guarantees for the quality of the data approximation. The AMS sketch defines the i -th sketch entry for the vector v , $sk(v)[i]$ as the random variable $\sum_k v[k] \cdot \xi_i[k]$, where $\{\xi_i\}$ is a family of four-wise independent binary random variables uniformly distributed in $\{-1, +1\}$ (with mutually-independent families across different entries of the sketch). Using appropriate pseudorandom hash functions, each such family can be efficiently constructed on-line in logarithmic space. Note that, by construction, each entry of $sk(v)$ is essentially a randomized linear projection (i.e., an inner product) of the v vector (using the corresponding ξ family), that can be easily maintained (using a simple counter) over the input update stream. Every time a new stream element streams in, we just have to add $v[k] \cdot \xi_i[k]$ to the aforementioned sum and similarly for element deletion – expiration. Each sketch vector can be viewed as a two-dimensional $n \times m$ array (Buckets and Depth in Table 2), where $n = O(\frac{1}{\epsilon^2})$, and $m = O(\log \frac{1}{\delta})$, with $\epsilon, 1 - \delta$ being the desired bounds on error and probabilistic confidence, correspondingly. The “inner product” in the sketch-vector space for both the join and self-join case (in which case we replace $sk(v_2)$ with $sk(v_1)$ in the formula below) is defined as:

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

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

Goal: Build small-space summary for distribution vector $f(i)$ ($i=0, \dots, N-1$) seen as a stream of i -values



Over the stream, add ξ_i whenever the i -th value is seen:



Figure 16: AMS Sketch computation example [19].

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

5.3.8 Chain Sampling

The chain sampling algorithm [25] provides a simple random sample without replacement of size k over a sliding window of n cardinality where k is expected to be much lower than n . We here describe the algorithm for sampling a single element from the sliding window. In order to obtain a sample of k size, this process needs to be repeated k times for each element.

 Project supported by the European Commission Contract no. 825070	<h2>WP6 T6.1</h2> <h2>Deliverable D6.1</h2>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public

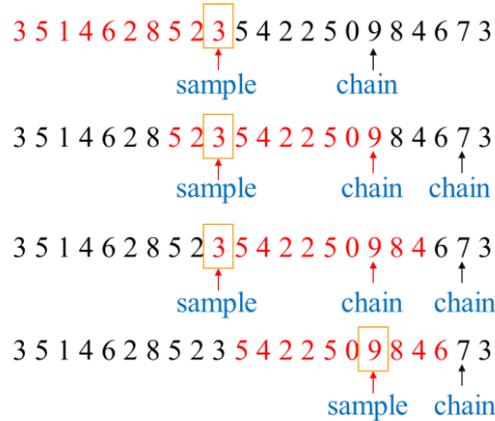


Figure 17: Chain Sampling example.

The chain sampling algorithm works as follows:

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

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

5.3.9 GK Quantiles

The GK algorithm [26] maintains a quantile summary Q as a collection of s tuples t_0, t_2, \dots, t_{s-1} where each tuple t_i is a triplet (v_i, g_i, Δ_i) : (i) a value v_i that is an element of the ordered version of the incoming stream (set) S (ii) the value g_i is equal to $\text{rmin}_{GK}(v_i) - \text{rmin}_{GK}(v_{i-1})$ (for $i = 0, g_i = 0$) and (iii) the value Δ_i which equals $\text{rmax}_{GK}(v_i) - \text{rmin}_{GK}(v_i)$. In general, $\text{rmin}(v)$ corresponds to a lower bound on the rank of v (whatever is included in the parenthesis), while $\text{rmax}(v)$ is an upper bound on the rank of v in S . The elements v_0, v_1, \dots, v_{s-1} are in ascending order, v_0 is the minimum element in S and v_{s-1} is the maximum element in S . Note that $n = \sum_{j=1}^{s-1} g_j$ which is the number of elements seen so far in the stream and $\text{rmin}_{GK}(v_i) = \sum_{j \leq i} g_j, \text{rmax}_{GK}(v_i) = \Delta_i + \sum_{j \leq i} g_j$.

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

Then a merging operation follows: we view the number of elements that have been observed over the incoming stream as an indication of time. Suppose that v in $t = (v, l, \Delta)$ arrives at n' and is placed in Q according to the above procedure with $\Delta \approx 2\epsilon n'$. At $n > n'$, we term the capacity of the tuple t as $2\epsilon n - \Delta$. As n is becoming higher with the arrival of new stream elements, the capacity of the tuple t increases as we can have more error (Δ remains steady, but $2\epsilon n$ strictly increases with n). We can thus merge tuples t_1, t_{i+1}, \dots, t_i into a single tuple t_{i+1} at (time) n with precision $\sum_{j=l}^{i+1} g_j + \Delta_{i+1} \leq 2\epsilon n, g_{i+1} = \sum_{j=l}^{i+1} g_j$.

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

 Project supported by the European Commission Contract no. 825070	<h2>WP6 T6.1</h2> <h2>Deliverable D6.1</h2>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public

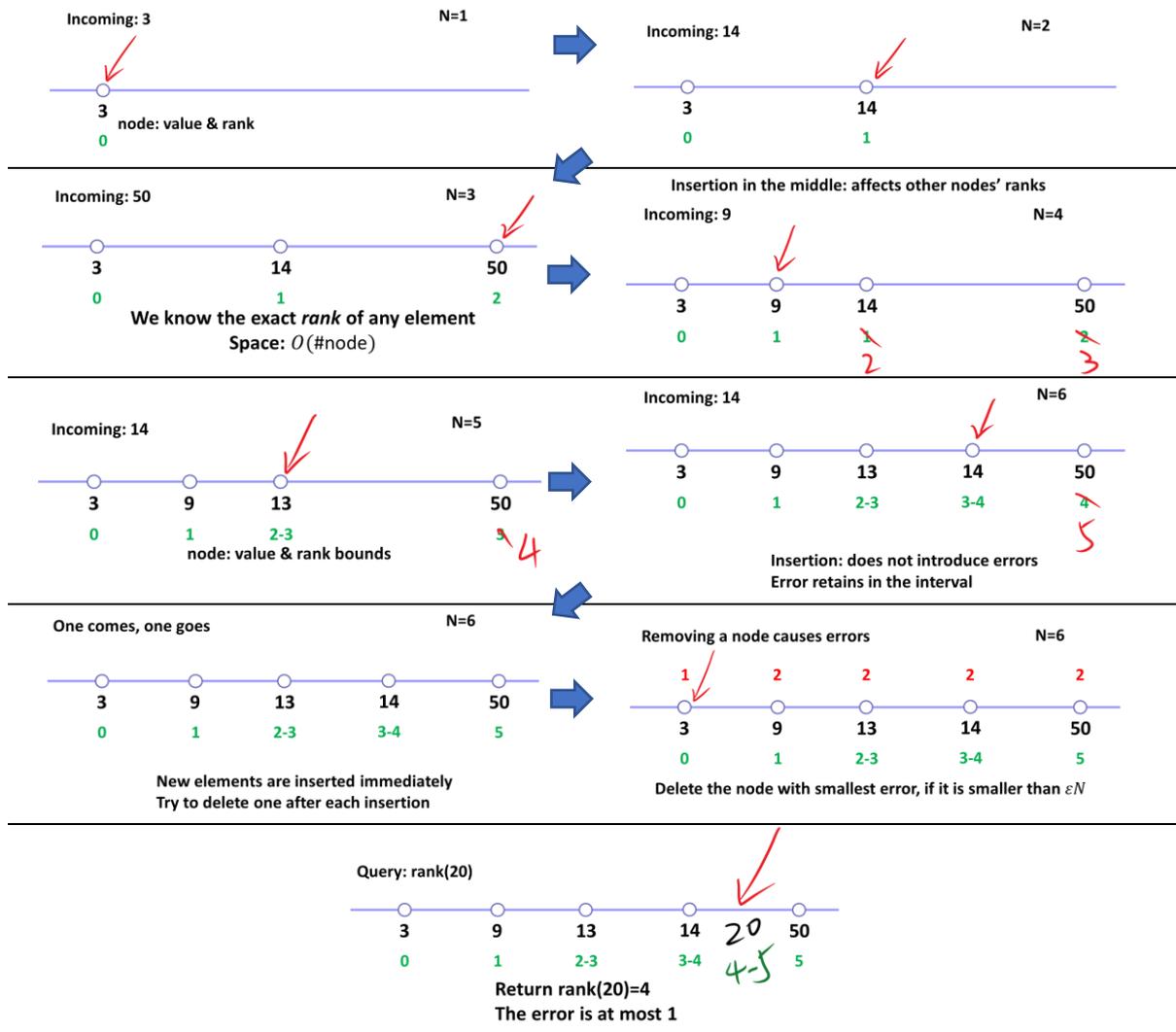


Figure 18: GK Quantile – Progressive insert and merge, query operation example [27]²⁶.

²⁶ <https://www.slideshare.net/jins0618/ke-yi-small-summaries-for-big-data>

<p>Project supported by the European Commission Contract no. 825070</p>	<h2>WP6 T6.1</h2> <h3>Deliverable D6.1</h3>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public

6 Performance Evaluation

To test the performance of our SDE, we utilize a Kafka cluster with 3 Dell PowerEdge R320 Intel Xeon E5-2430 v2 2.50GHz machines with 32GB RAM, 1TB HDD each and one Dell PowerEdge R310 Quad Core Xeon X3440 2.53GHz machine with 16GB RAM, 500GB HDD. Our Flink cluster has 10 Dell PowerEdge R300 Quad Core Xeon X3323 2.5GHz machines with 8GB RAM, 500GB HDD each.

We use a dataset provided by Spring composed of ~5000 stocks contributing a total of 2.6 Billion trades (updates), which averages to 500K updates per stock. In our experiments we measure the throughput, expressed as the number of tuples being processed per time unit (second), while varying a number of parameters involving:

- the number of summarized stocks (streams) [50-500-5000] with a default value of 500,
- 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 with a default value of 10×
- the parallelization degree [2-4-6-8-10] with a default value of 4.

In each experiment, we build and maintain Discrete Fourier Transform (DFT), HyperLogLog (HLL) and CountMin (CM) synopses each of which is destined to support different types of analytics related to correlation, distinct count and frequency estimation, respectively (Table 2).

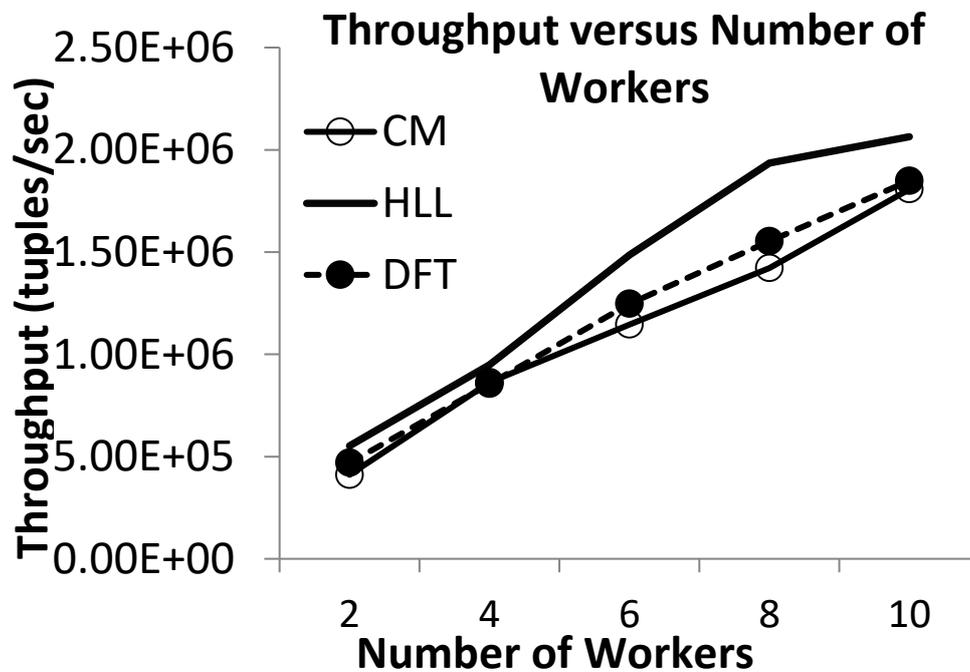


Figure 19: Scalability evaluation varying the parallelization degree

Figure 19 shows that increasing the number of Flink workers causes an analogous 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.

 Project supported by the European Commission Contract no. 825070	<h3>WP6 T6.1</h3> <h2>Deliverable D6.1</h2>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public

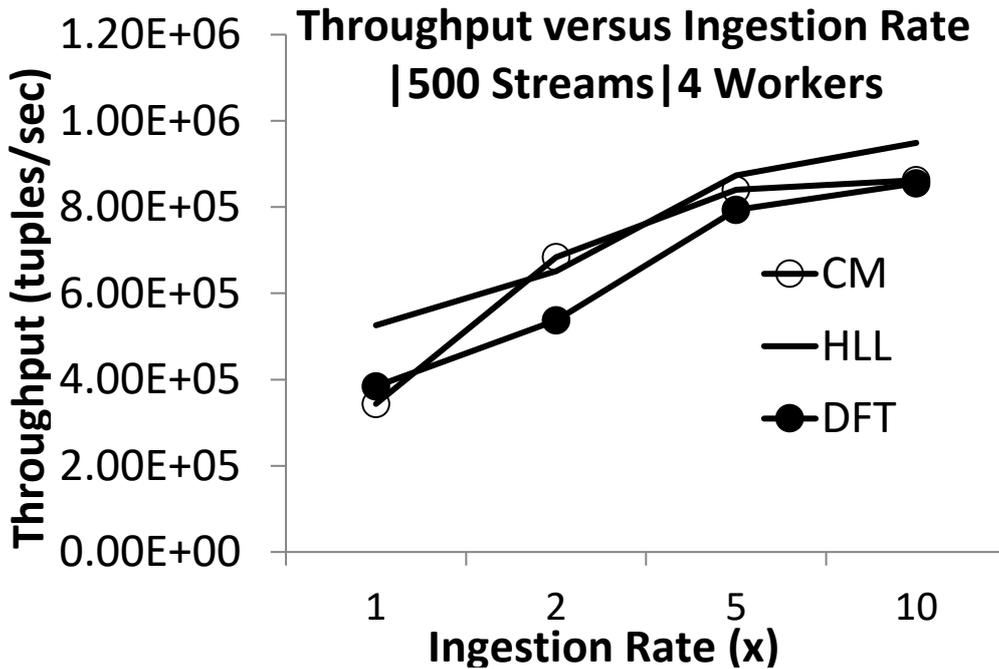


Figure 20: Scalability evaluation varying the ingestion rate

Figure 20, 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 SDE can serve, quantified in terms of throughput, are equivalent to the rates at which data arrive to it.

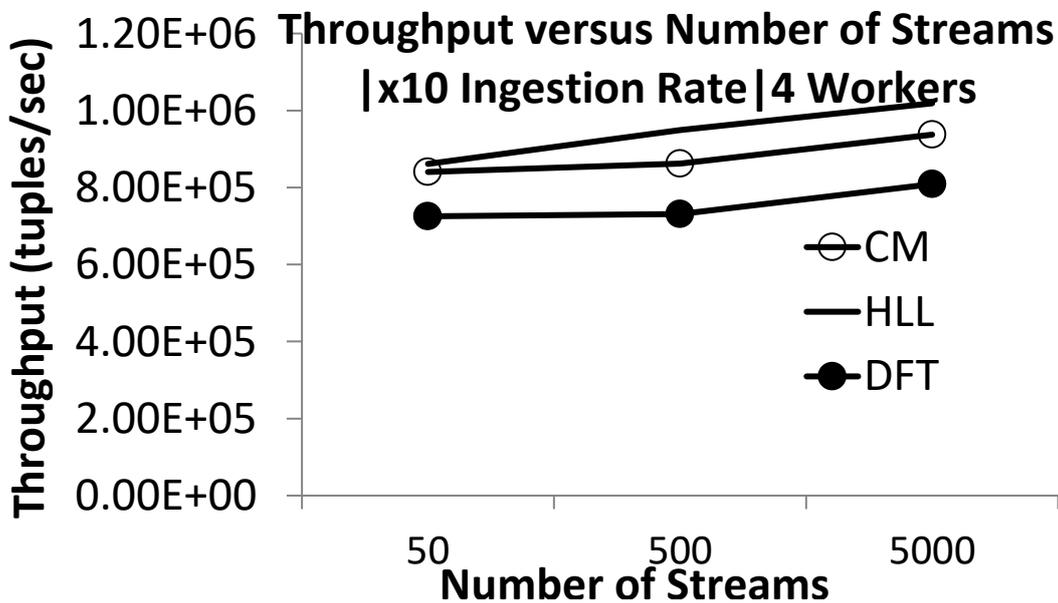


Figure 21: Scalability evaluation varying the number of monitored streams

Finally, Figure 21 shows that throughput is not severely affected upon increasing the number of processed streams from 50 to 5000 which validates our claim regarding the vertical scalability aspects the SDE can bring in the workflows it participates.

<p>Project supported by the European Commission Contract no. 825070</p>	<p>WP6 T6.1 Deliverable D6.1</p>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public



7 Conclusions and Future Work

This deliverable detailed the design choices and the internals of the first version of the Data Synopsis Generator of INFORE, which we term as Synopses Data Engine (SDE). We discussed the non-functional requirements that led us implementing a proof-of-concept of the SDE on Apache Flink and Apache Kafka. We outlined the API the SDE provides to upstream and downstream operators and we elaborated on the design choices and the internals of the SDE architecture. We further described the structure of the SDE library and presented the functionality of the currently incorporated synopses. At its current stage the SDE library includes both synopses destined to serve INFORE use case needs, as well as synopses that have been proven popular and useful to a wide variety of application scenarios. Finally, we presented experimental results showing the scalability of the SDE component under varying ingestion rate, number of concurrently processed streams and parallelization degree.

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

- **Dynamic loading of synopses from external library:** a workflow should be allowed to incorporate any synopsis operator that abides by the SDE’s library structure (see Figure 5) on the fly without needing to shut down the SDE component and restart it afterwards. The latter approach would stall the execution of all currently executed workflows that utilize the SDE and is therefore unacceptable for INFORE’s online analysis needs.
- **Incorporation of more synopsis techniques:** the SDE library will be enhanced with a number of additional data summarization techniques that match the needs of a wide variety of applications. Carefully inspecting Table 2, one can easily deduce that at the current stage of the SDE library we have chosen, besides synopsis tailored to INFORE use cases, at least one techniques for (distinct) count, frequency, set membership, cardinality, correlation and sample estimation. These constitute broad categories of synopsis often used in various application scenarios. This list is going to be enhanced with other popular, approximate operators involving approximate top-K query answering and histogram computation. Moreover, more than one algorithm will be incorporated per category based on criteria and the variety of trade-offs they introduce compared to the already incorporated ones. Such criteria are related to mergeability, memory utilization, computational complexity of updating and querying the synopsis.
- **Joins & Extended Windowing Support:** the synopses we have currently incorporated, in large part operate over the whole stream history or disjoint windows of streams. In the future we plan to incorporate synopses that can equivalently support both tumbling (disjoint) and sliding windows. Moreover, we have designed a way that the current architecture could be used so as to implement the functionality of popular approximate join algorithms.
- **Federated Scalability:** our experimental evaluation is focused on horizontal (voluminous, high speed data streams), and vertical (high number of concurrently processed streams) scalability on a single data summarization technique. In the future we intend to expand our experimental endeavors to (a) perform comparisons of synopses destined to approximate equivalent quantities, (b) assess federated scalability upon operating in multi-cluster environments.
- **Benchmarking:** One of the main advantages of the SDE, besides providing techniques for reduced memory utilization, is that it can work on small, carefully-crafted portions (such as samples) of the data and provide rapid responses within the scope of complex workflows of Big Data analytics. This is true in the Life Science use case were the SDE, and the DFT or LSH techniques in particular, can help to rapidly detect useful, correlated simulations or non-useful, correlated ones so as to provision less cluster resources for the latter category and more resources for the former. It is even true in the Financial use case for equivalent, stock similarity computations within the scope of system risk prediction, or in the Maritime use case where proximity events need to be detected among every possible pair of vessels sailing within a certain region. Measuring and predicting the speed up an approximate operator of the SDE can provide for the workflow execution, compared to the equivalent exact operator can help us utilize this knowledge to the INFORE optimizer in order to: (i) perform corresponding suggestions to those designing a visual workflow in the INFORE’s design tool, i.e., that by substituting a workflow operator with an equivalent, approximate one present in the SDE library a certain execution speed up is expected and (ii) incorporate benchmarking results to perform synopsis-based optimization [28][29] within the scope of the INFORE optimizer. That is, the optimizer may then have the opportunity, given an accuracy budget the application prescribing the workflow is willing to spend, to examine alternative workflow execution plans by substituting exact operators with approximate ones and estimating the expected speed up.

 <p>Project supported by the European Commission Contract no. 825070</p>	<h3>WP6 T6.1</h3> <h2>Deliverable D6.1</h2>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public

8 References

- [1] Nikos Giatrakos, Nikos Katzouris, Antonios Deligiannakis, Alexander Artikis, Minos N. Garofalakis, George Paliouras, Holger Arndt, Raffaele Grasso, Ralf Klinkenberg, Miguel Ponce de Leon, Gian Gaetano Tartaglia, Alfonso Valencia, Dimitrios Zissis: Interactive Extreme: Scale Analytics Towards Battling Cancer. *IEEE Technol. Soc. Mag.* 38(2): 54-61 (2019)
- [2] N. Giatrakos, E. Alevizos, A. Artikis, A. Deligiannakis, M. Garofalakis: "Complex Event Recognition in the Big Data Era: A Survey", *The VLDB Journal*, 2019, <https://doi.org/10.1007/s00778-019-00557-w>.
- [3] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, Volker Markl: Benchmarking Distributed Stream Data Processing Systems. *ICDE 2018*: 1507-1518.
- [4] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of NetDB*, 2011.
- [5] Bonaventura Del Monte, Jeyhun Karimov, Alireza Rezaei Mahdiraji, Tilmann Rabl, Volker Markl: PROTEUS: Scalable Online Machine Learning for Predictive Analytics and Real-Time Interactive Visualization. *EDBT/ICDT Workshops 2017*.
- [6] M. Garofalakis, J. Gehrke, and R. Rastogi, editors. *Data Stream Management - Processing High-Speed Data Streams. Data-Centric Systems and Applications*. Springer, 2016.
- [7] P. K. Agarwal, G. Cormode, Z. Huang, J. Phillips, Z. Wei, and K. Yi. Mergeable summaries. In *Proceedings of PODS*, 2012.
- [8] G. Cormode, M. Garofalakis, P.J. Haas, and C.M. Jermaine. "Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches", *Foundations and Trends in Databases*, Vol. 4, No. 1-3, pp. 1-294, 2012. (ISBN 978-1601985163).
- [9] N. Giatrakos, Y. Kotidis, A. Deligiannakis, V. Vassalos, Y. Theodoridis: TACO: tunable approximate computation of outliers in wireless sensor networks. *SIGMOD Conference 2010*: 279-290
- [10] N. Giatrakos, Y. Kotidis, A. Deligiannakis, V. Vassalos, Y. Theodoridis: In-network approximate computation of outliers with quality guarantees. *Inf. Syst.* 38(8): 1285-1308 (2013).
- [11] Konstantinos Georgoulas, Yannis Kotidis: Distributed similarity estimation using derived dimensions. *VLDB J.* 21(1): 25-50 (2012)
- [12] Yunyue Zhu, Dennis E. Shasha: StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. *VLDB 2002*: 358-369
- [13] Nikolaos Pavlakis, Scaling out streaming time series analytics on Storm, MSc Thesis, Technical University of Crete, 2017, <http://purl.tuc.gr/dl/dias/802769EF-66BD-411D-9E6E-BA00A61F2B6F>
- [14] Rakesh Agrawal, Christos Faloutsos, Arun N. Swami: Efficient Similarity Search In Sequence Databases. *FODO 1993*: 69-84
- [15] M. Potamias, K. Patroumpas, and T. Sellis. Sampling Trajectory Streams with Spatiotemporal Criteria. In *Proceedings of the 18th International Conference on Scientific and Statistical Database Management (SSDBM '06)*, 2006.
- [16] G. Cormode, S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms* 55(1): 58-75 (2005).
- [17] G.S. Manku, R. Motwani, Approximate frequency counts over data streams, in: *VLDB, 2002*, pp. 346–357.
- [18] Graham Cormode and Minos Garofalakis. "Streaming in a Connected World: Querying and Tracking Distributed Data Streams" (tutorial abstract), *Proceedings of VLDB'2006*, 2006.
- [19] Minos N. Garofalakis, Johannes Gehrke, and Rajeev Rastogi. "Querying and Mining Data Streams: You Only Get One Look" (tutorial abstract), *Proceedings of ACM SIGMOD*, 2002.
- [20] P. Flajolet and G. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182-209, 1985.
- [21] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 1970, 422-426. DOI: <https://doi.org/10.1145/362686.362692>
- [22] P. Flajolet, Éric Fusy, O. Gandouet, and F. Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *Analysis of Algorithms (AOFA)*, pages 127--146, 2007.
- [23] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In G. D. Battista and U. Zwick, editors, *European Symposium on Algorithms (ESA)*, volume 2832, pages 605--617, 2003.
- [24] N. Alon, Y. Matias, and M. Szegedy. "The Space Complexity of Approximating the Frequency Moments". In *ACM STOC*, 1996.

 <p>Project supported by the European Commission Contract no. 825070</p>	<h3>WP6 T6.1</h3> <h2>Deliverable D6.1</h2>	Doc.nr.:	WP6 D6.1
		Rev.:	1.0
		Date:	20/12/2019
		Class.:	Public



- [25] Brian Babcock, Mayur Datar, and Rajeev Motwani. 2002. Sampling from a moving window over streaming data. In Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA '02). 633-634.
- [26] Michael Greenwald, Sanjeev Khanna: Space-Efficient Online Computation of Quantile Summaries. SIGMOD Conference 2001: 58-66.
- [27] Graham Cormode and Ke Yi: Small Summaries for Big Data, Capt. 4, Cambridge University Press, 2020, <http://dimacs.rutgers.edu/~graham/ssbd.html>
- [28] K. Li, G. Li. Approximate Query Processing: What is New and Where to Go? Data Science Engineering, 2018. <https://doi.org/10.1007/s41019-018-0074-4>
- [29] S. Nirkhiwale, A. Dobra, and C. Jermaine. A sampling algebra for aggregate estimation. In Proc. VLDB Endow. 6, 14, 2013, 1798-1809.

 European Commission Horizon 2020 European Union Funding for Research & Innovation	Project supported by the European Commission Contract no. 825070	<h2>WP6 T6.1</h2> <h3>Deliverable D6.1</h3>	Doc.nr.: WP6 D6.1
			Rev.: 1.0
			Date: 20/12/2019
			Class.: Public