Technical University of Crete

School of Electrical & Computer Engineering

Diploma Thesis

Persistency and Interoperability in Synopses Data Engine: Integration into Knowledge Lakes

This Thesis Was Submitted in Partial Fulfillment of the Requirements
for the Diploma in Electrical and Computer Engineering

Author:

Dimitrios G. Petrou

Thesis Committee:

Prof. Vasileios Samoladas, *Supervisor*

Prof. Antonios Deligiannakis

Prof. Nikolaos Giatrakos

February 2025

Abstract

In the era of big data where real-time information is generated at an unprecedented scale, the ability to process, analyze, and extract actionable insights efficiently is considered a requirement in many use cases. Stream summarization has emerged as novel technique when it comes to addressing this, enabling the creation of compact, yet informative, representations of continuous data streams, termed synopses, eliminating the need of storing vast amounts of raw data for future processing. A prominent effort conducted in this sector, is the Synopses Data Engine (SDE), an advanced framework that integrates state-of-the-art stream summarization techniques with the high-performance capabilities of Apache Flink, eventually forming an interactive summarization service at a scale. While SDE has proven its merit in other big data ecosystems, its application within knowledge-driven environments introduces new requirements. In their native form, synopses are volatile. Their lifespan depends on the runtime of the engine. However, in the context of Knowledge Lakes, where long-term insights and temporal analytics are essential, the inability to retain and revisit previous states introduces a significant limitation. This thesis aims to bridge this gap by extending the capabilities of SDE to incorporate persistency and a versatile snapshot mechanism, allowing for the long-term storage and retrieval of Synopses by respecting SDE's indigenous key features. Furthermore, the work expands the Streaming API of SDE to provide broader observability into the internal state of the engine, allowing metadata to be extracted towards outer data analytics ecosystems. The STELAR KLMS (Knowledge Lake Management System) serves as the domain of application for this Thesis, where SDE is integrated to process real-time agri-food data for precision interventions.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor and mentor, Professor Vasilis Samoladas for his unwavering support, insightful guidance along this journey, and opportunities he has exposed me to. Throughout his course on *Distributed Systems*, I have not only acquired extensive knowledge in the field but also found continuous inspiration far beyond it. His passion on explaining concepts and willingness for passing on further understanding led my academic evolution up to the completion of this Thesis.

My gratitude extends to the members of the committee of this Thesis, Professor Antonios Deligiannakis and Professor Nikos Giatrakos, for their valuable insights on my work and their previous contributions in the Synopses Data Engine. Throughout their courses I was introduced to concepts that grasped my interest and formed who I am today as a future engineer.

I would like to also acknowledge the unconditional support I've received from Nikos Bakatselos and Michael Theologitis in our every day interactions and year-long collaboration, during our participation in the research team of the SoftNET lab at Technical University of Crete. My appreciation extends to the entire team participating in the STELAR project and especially to Wieger R. Punter for his valuable contributions complementing my work.

Last but not least, I am immensly grateful to my family and friends, for making this journey a unique experience and from who, I am constantly receiving invaluable and unconditional support.

This thesis was partially funded by the European Commission under the STELAR (HORIZON-EUROPE - Grant Agreement No. 101070122) project.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

In recent research efforts, stream summarization begans to emerge as a critical technique for handling large-scale, high-velocity data streams in an efficient way. The mobility observed in this research area is a result of the ever growing volume of real-time data in the digital world today and the need for efficient processing on them. Real-time data often are unbounded, and retaining them in storage seams impractical or most of times impossible as the storage needs will be ranging from petabytes to exabytes in a matter of days. Summarization techniques focus on creating compact, yet comprehensive and informative structures, termed as *Data Summaries* on continuous influxes of data by having access to the original information only once, eliminating this way the need for retaining, in storage mediums, the entire volume of streams.

Summaries, hereinafter termed as *Synopses*, have small memory footprint and are capable of providing statistical insights on ingested data with low error bound guarantees on estimations. Novel works [1] have been introduced in latest years regarding the theoretical and algorithmic background of *Synopses*, while others focus on the functional aspect of them aiming to accomodate them as parts of an analytics framework [2]. One of the most prominent and complete works that were elaborated in recent years was the *Synopses Data Engine* [3], which accomodates *Synopses* as first-class citizens wrapping them in framework where interactive stream summarization is provided as a service (*SDEaaS*). *SDE* leverages the powerful *DataStream API* of *Apache Flink* [4] to built a versatile *Dataflow* for implementing an engine for maintaining, updating and querying *Synopses* on the fly. The engine lays its foundation in the virtues of *Flink's* parallel processing capabilities and data summarization most novel techniques to eventually provide an integrated and interactive summarization framework at a scale.

1.2 Problem Statement

SDE has succefully been integrated before as part of big data ecosystems, providing data analysts with a flexible tool that generates insights on unbounded ingested in real-time *Data Streams* or static datasets.

Taking into account the efficiency and the range use-cases *Synopses Data Engine* may find application in, we envisioned its integration as a part in the toolkit of the *STELAR KLMS* (Knowledge Lake Management System) ecosystem. *STELAR* [5] aims

to design and develop a KLMS for turning raw data lakes into knowledge lakes focusing mainly in the Agri-Food domain. Data analytics and interlinking tools are encompassed in *STELAR's* agenda, facilitating this way the discovery, reusability and interoperability of data. *SDE* comes in handy for providing actionable, real time and fast insights for a range of applications in *STELAR*, where the raw-data may be oftenly utilized in the form of streams. Estimations and states produced by the engine need to be tracked and retained thoroughly, in order for them to form generated knowledge alongside other statistics incoming from the lake's components.

SDE supports *Synopses* from the bottom up in an integrated way, but *Synopses* have a highly-volatile nature, meaning that their lifespan is limited to the engine's runtime, while historical observability of states they have been through is not currently facilitated. In the context of a *Knowledge Lake*, for which *SDE* is accounted in this study, temporal and historic statistics or insights and rollback capabilities are considered a non-negotiable requirement.

In this work, driven by the aforementioned fact, we expand the capabilities of the *Synopses Data Engine* in terms of stateful processing by designing and implementing features that incorporate persistency on running *Synopses* with a versatile snapshot mechanism. *Snapshots* may be used in the context of the lake, to be loaded back in the engine as *Synopses*, making them recoverable on every captured past state. Additionally, the focus of our work expands to enriching the *Streaming API* offered by the engine to further support the observability of the internal state of the engine in terms of metadata, marking it as interoperable component. Metadata extracted from the engine's internal world, can be later incorporated as additional insights in any software ecosystem.

1.3 Thesis Outline

This Thesis consists of 4 chapters, excluding the current one, these are:

2. Related Work. In this chapter the foundational concepts that orient this Thesis are laid. Similar research efforts are outlined and assessed thoroughly, while concepts that are adjacent to the next chapters are showcased, providing a comprehensive knowledge framework that will help address the contributions of this work.

3. Persistency in Sde. A detailed survey is conducted on the design that was applied to the *SDE*, aiming in order for it to accomodate the new envisioned features for persistency and snapshotting of Synopses. Concepts ranging from the selected tools and techniques to the final implementation are exhaustively elaborated.

4. Integration. This chapter delivers an outline, both in terms of implementation and design, on the path that was paved to integrate *SDE* into a *Knowledge Lake* as part of its toolkit. We explain, how the need for metadata within the lake enabled us to further expand the capabilities of *SDE*. Last, but not least we showcase the techniques used to integrate the engine within the lake's ecosystem while addressing various challenges that emerged on the way.

5. Conclusions. We conclude the current work assessing its contribution in both the *Synopsis Data Engine* as a standalone component and as a part of greater data analytics ecosystem. Objectives for future work are also included.

Chapter 2

Related Work

2.1 Data Streams

In this section we delve into the fundamentals of data streams and their supplementary topics which will help us address core parts of this thesis. The concept of a "Data Stream" is addressed in relation to software engineering and data analysis, as the term is shared with other fields of computer engineering. (Operating Systems, Computer Architecture, etc.). We will address the concept both structurally and functionally.

2.1.1 What Is a Data Stream?

In software engineering, data streams are defined as series of data elements (numeric, text or more complex-structured) that are provided or generated gradually over time, usually in real-time. They indicate an uninterrupted stream of information that can be handled, processed online, or sent instantly. Unlike static datasets, data streams are dynamic, often unbounded. It is not uncommon, that data or events in streams are timewise sorted. Data streams hold a crucial role in multiple scenarios, ranging from instant analytics to decision-making, and serve as a key principle in systems for managing and processing real-world events.

Data Streams occur, more often than not, in our everyday life. Credit card transactions, cloud service access records, stock price variations and more scenarios can result in the formulation of a data stream of events or data, which can be subsequently driven into a processing software module for further analysis.

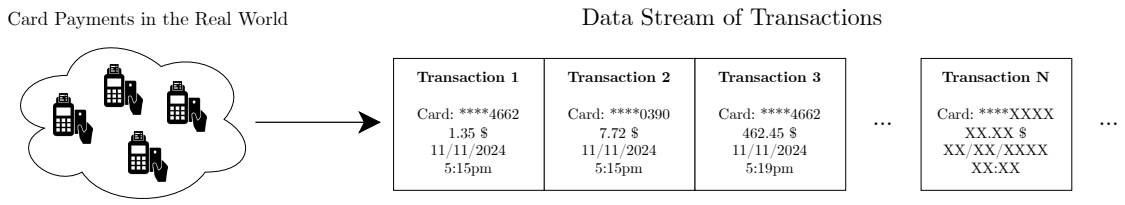


Figure 2.1: Example data stream created upon credit card payment events in real world. Each payment represents a transaction item in the stream.

Further examples can be fashioned upon other scenarios. We will use the one above, for simplicity's sake, wherever needed in the next parts of this section.

2.1.2 Data Stream Generation

Data streams can be generated or formulated from various producers or sources:

1. Real-Time Producers. Sensors, IoT devices, GPS trackers, Maritime Radars, Finance Market Data (Prices, Trades)

2. Message Brokers. Message brokers can ingest messages from a source and digest them as a message stream

3. Data Streaming Platforms. Frameworks or software like *Apache Flink* [4], are able to construct data streams based on database records, data structures or even plain text files.

4. Generic Sources. Every generic source that can read or produce, format and digest data into a continuous stream.

While there seems to be a bounded categorization of sources, please bear in mind that unlimited possibilities exist when discussing what can be qualified as a "Data Stream" source.

2.1.3 Data Stream Processing

The processing of one or more data streams can be addressed in both a systematic and a conceptual form. Systematically-wise, an efficient way of managing streams is deemed necessary while the sensibility and integrity of data must be guaranteed. This is not a task that can be addressed and worked out in the blink of an eye. More and more complications occur when someone tries to handle streams from scratch. Fortunately, a variety of frameworks and services are open to the public today, serving as data stream management platforms with more than adequate features.

On an abstract point of view, Kontaxakis (2020) [6] compares a *Database Management System* (DBMS) with a *Data Stream Management System* (DSMS). Although such systems serve divergent purposes, they can be assessed based on their operation principle. As stated the key difference among the two schemes is the path followed by queries and data. Given a DBMS, the data are stored permanently on disk and the system is responsible for handling incoming queries, executing required operations on data and extracting the result of the query. On the other hand, a DSMS will not accept queries as input. In the DSMS case, the system ingests data (formed of data streams) that affect a running query and a result is digested in form of a stream. A more clear explanation of this point can be given by the below diagram:

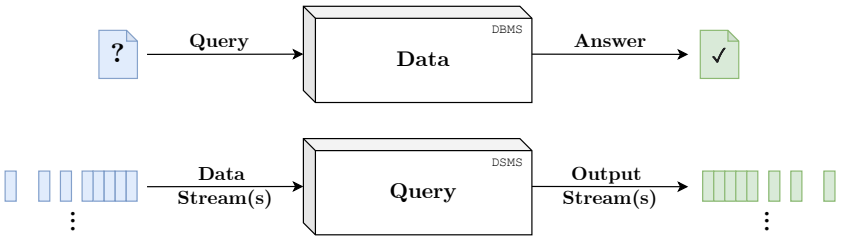


Figure 2.2: *Database Management System* VS *Data Stream Management System*

From a technical scope, some evaluation points may be established in order to define what is considered a reliable and handy DSMS:

1. Latency & Throughput. Low latency is critical for real-time applications while high throughput is essential for handling large-scale data streams
2. Query Language & Expressiveness. Ease of use, complexity, and capability of the query language to express complex data processing tasks
3. Scalability. Both vertical & horizontal scalability. Scaling in resources or scaling computerwise
4. Universality. Connectivity features to third party services
5. Fault Tolerance. Fault recovery mechanisms and fault prevention features

Undoubtedly, the criteria above are not fixed or definitive, as specific use cases may introduce new ones or place greater emphasis on certain aspects of the system.

Conversely, it is crucial that the processing of one or more data streams yields sensible and useful results. This is reliant upon both the design and capabilities of the Data Stream Management System (DSMS) as well as the design and implementation quality of the processing workflow by the developer. The broader context of this field is known as data stream mining.

2.1.4 Data Stream Mining

The majority of streams do not have a story to tell on their own. Appropriate, use-case dependent processing and analysis, should take place in order to transform raw data into actionable insights that can inform decision-making processes. Data mining involves the application of algorithms and techniques to discover hidden patterns within data. Techniques such as regression, classification, aggregation and more are employed for this purpose.

While Data Mining is a multidisciplinary field, drawing on elements of statistics, computer science, and artificial intelligence to address complex problems, appropriate tools should exist in order to incorporate the complicated processes required. The quality of those tools lies on the key evaluation criteria mentioned in the previous part, while specific scenarios may also modify or fine-tune them.

2.2 Apache Flink

Apache Flink belongs to the broader family of *Data Stream Management Systems (DSMS)*, offering a platform and an API for stream and batch processing. Flink introduces several novel features that align with and fulfill the majority of the quality criteria that were previously outlined:

- Real-Time Processing. Flink processes data as it arrives, enabling near-instant insights and responses to events.
- Rich Api. Flink's DataStream API covers a vast amount of operations and processing techniques that can be applied to streams or batches.
- Parallelization. Accomplished by partitioning data and distributing tasks across independent task slots in a cluster.
- Connectivity. Flink can ingest and digest streaming data from/to various sources. The Flink API offers a library of prebuilt connectors that cover the most known Messaging Brokers, Databases and Data Warehouses. The library also contains more generic connectors, both for input and output.
- Fault Tolerance. Flink ensures reliability with features like state snapshotting and recovery mechanisms. If a failure occurs, it can recover processing state and resume without losing data.

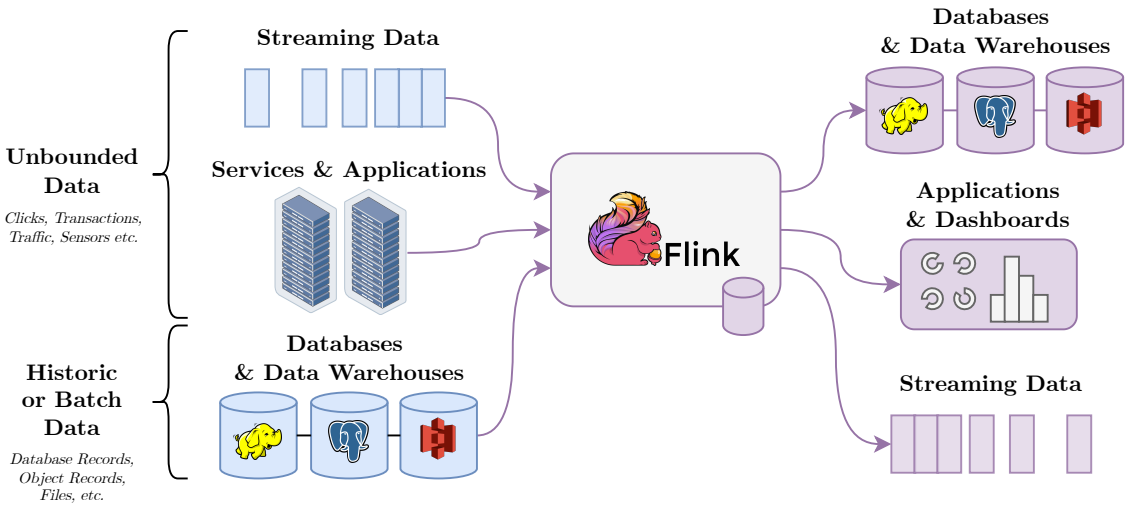


Figure 2.3: *Apache Flink* abstract architecture with sources and sinks.

2.2.1 System Architecture

Flink's architecture follows a master-worker model, where the JobManager (mast-er) coordinates job execution by receiving job submissions, building a logical plan, and transforming it into an optimized execution graph. Additionally it is responsible for scheduling tasks and allocating resources for them in the cluster. On the other hand, TaskManagers (workers) run the actual tasks, they leverage the JVM for efficient memory management

(heap, off-heap, and network buffers) and thread isolation, ensuring optimized task execution and communication. link also provides robust fault-tolerance features, such as distributed snapshots and checkpointing, allowing jobs to recover from failures without losing state across the cluster.

2.2.2 Dataflows

Conceptually, a program written to be executed on a *Flink Cluster* [1] is a directed acyclic graph (DAG) of operators propagating one or more data streams through them. A Flink *Dataflow* is a well defined and finite sequence of operations that are applied to one or more initial streams of data. The initial streams can be either bounded or unbounded and the source of them can be a variety of producers. As an open-source project, Flink offers a great set of connectors for ingesting data like Apache Kafka, Amazon S3, Databases (SQL and Document) and more.

As stream data flow through the chain of operators, they undergo processing until a final result is produced according to the dataflow design. The final result can be a stream or a batch of data, depending on the nature of the operations applied. Output data are digested through sinks, which are the final nodes in a dataflow and can write processed results to various external systems (e.g., files, databases, or message brokers).

Conclusively, Flink's dataflow model is a powerful abstraction that makes Flink a versatile DSMS for complex stream analytics. Below is a simple example of how

2.2.3 DataStream API Fundamentals

DataStream API is the core interaction point with Flink. As a high level interface, it provides the set of operators that can be used to build complex data processing pipelines. The API is designed to be expressive and flexible, allowing developers to define even custom operators and transformations deriving from built-in ones. It is compatible with most modern programming languages, including Java, Scala, and Python, each supported by dedicated Flink libraries, thereby enabling seamless development of dataflows across diverse computational paradigms.

Data is often represented in a structured format for processing within the Flink's context. The most usual and convenient way to data is the *DataStream* object, which acts as a collection of events, without limitations on bounds. A DataStream is similar to a regular Java Collection in terms of usage but is quite different in some key ways. Most of them can be represented as an ordered sequence of data elements:

$$S = \{e_1, e_2, e_3, ...\} \tag{2.1}$$

$$|\mathbf{S}| < \infty \text{ for } \textit{bounded streams}, |\mathbf{S}| \to \infty \text{ for } \textit{unbounded streams}$$

Where each element e_i is a tuple :

$$e_i = (k, v, t)$$

- **k**: Key. It is optional, used for grouping and partitioning.
- **v**: Value, The actual data object of any type.
- **t**: Timestamp of the event, which can be event time (t_e) or processing time (t_p)

[1] A distributed installation of Flink across multiple nodes, used to process and analyze large-scale data streams in parallel.

Classic Transformation Operators

When formed, *DataStreams* may undergo a series of transformations, typically applied to their individual elements. Operators [7] transform one or more *DataStreams* into a new *DataStream*. Flink Programs can combine multiple transformations into sophisticated dataflow topologies. In the subsequent parts of this section, we provide an overview of the most commonly used operators in Apache Flink for manipulating data streams.

- `Map`: Applies a function f to each element in the stream. Takes one element and produces one element.

$$T_{map}(S) = S' = f(e_i) \mid e_i \in S \tag{2.2}$$

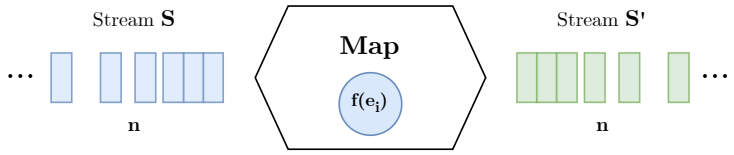


Figure 2.4: `Map` transformation of Flink's *DataStream* API

- `FlatMap`: Applies a function f to each element in the stream and produces zero or more elements. Takes one element and produces zero or more elements.

$$T_{flatMap}(S) = S' = f(e_i) = \{e_{i1}, e_{i2}, ..., e_{im}\} \mid \forall e_i \in S \exists e_{im}, 0 \leq m \leq \infty \tag{2.3}$$

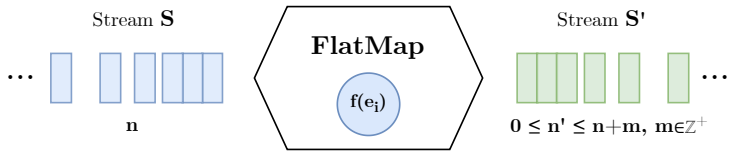


Figure 2.5: `FlatMap` transformation of Flink's *DataStream* API

- `Filter`: Applies a function f to each element in the stream and retains only those elements that satisfy a predicate p.

$$T_{filter}(S) = S' = f(e_i) = \{e_i \mid e_i \in S \wedge p(e_i) = \text{true}\} \tag{2.4}$$

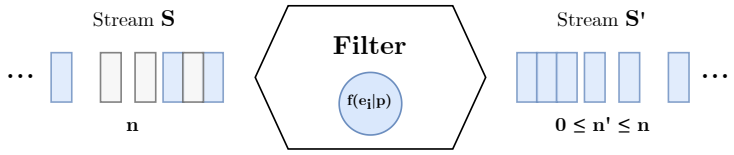


Figure 2.6: `Filter` transformation of Flink's *DataStream* API

- `KeyBy`: Logically partitions a stream into disjoint partitions based on a key k. Usually performed when a dataflow runs with parallelism greater than one. Commonly keyed elements are assigned to the same parallel task.

$$T_{keyBy}(S) = S' = \{S_1, S_2, ..., S_n\} \mid \forall e_i \in S, e_i \in S_j \wedge e_i \notin S_k, j \neq k \tag{2.5}$$

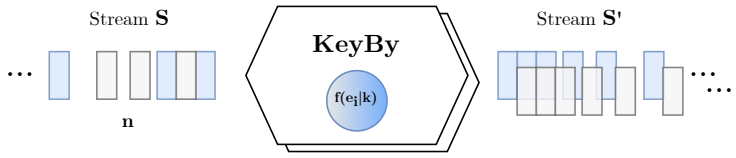


Figure 2.7: `KeyBy` transformation of Flink's *DataStream* API

- `Reduce`: Applies an associative and commutative function f to elements in the stream. It is used to aggregate elements from keyed streams to produce a single *DataStream*.

$$T_{reduce}(S) = S' = f(e_i, e_j) = e_k \mid e_k = f(e_i, e_j) \wedge e_i, e_j \in S \tag{2.6}$$

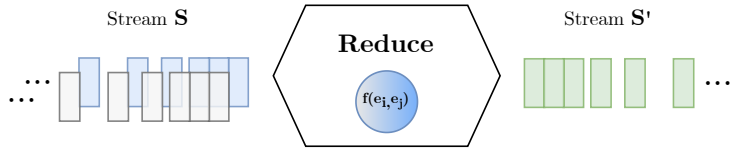


Figure 2.8: `Reduce` transformation of Flink's *DataStream* API

- `Union`: Merges two or more *DataStreams* into a single *DataStream*. The streams must have the same type.

$$T_{union}(S_1, S_2, ..., S_n) = S' = S_1 \cup S_2 \cup ... \cup S_n = \bigcup_{i=1}^{n} S_i \tag{2.7}$$

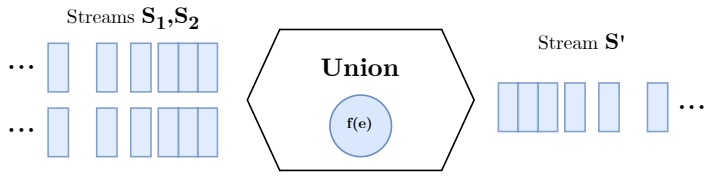


Figure 2.9: `Union` transformation of Flink's *DataStream* API

- `Connect`: Connects two *DataStreams* retaining their types. Allows feeding to shared state operators.

$$T_{connect}(S_1, S_2) = S' = \{e_1, e_2 \mid e_1 \in S_1, e_2 \in S_2\} \tag{2.8}$$

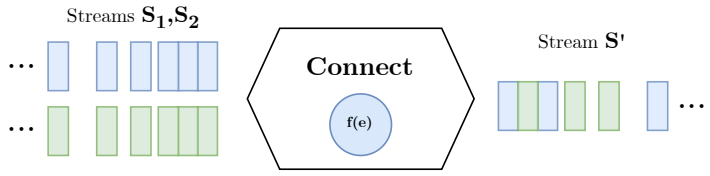


Figure 2.10: `Connect` transformation of Flink's *DataStream* API

Shared State Transformation Operators

It is commonly required to process multiple data streams concurrently, especially when the streams are interrelated or require joint analysis. Flink provides quite a diverse range of operators, designed to facilitate the processing of two connected streams in tandem. Computation takes place within the context of a *Shared State*, which serves as a mechanism to store intermediate computations, monitor pattern and share state across diversely typed streams. This approach is instrumental in enabling robust, stateful stream processing for use-cases demanding combined analysis of multiple streams.

- `CoFlatMap`: Applies a function to elements from two connected streams, allowing for the combination and transformation of elements from both streams.

$$T_{coFlatMap}(S_1, S_2) = S' = \{f(e_1, s), f(e_2, s) \mid e_1 \in S_1, e_2 \in S_2, s \in \text{state}\} \tag{2.9}$$

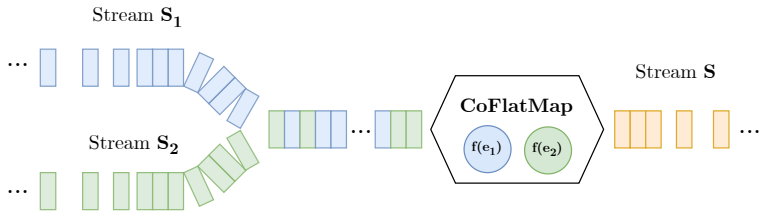


Figure 2.11: `CoFlatMap` transformation of Flink's *DataStream* API

- `CoProcessFunction`: Applies a function to elements from two connected streams, allowing for the combination and transformation of elements from both streams. In comparison with the `CoFlatMap` operator provides access to a versatile *context* with side output and temporal relationships between events.

$$T_{coProcess}(S_1, S_2) = S' = \{f(e_1, s), f(e_2, s) \mid e_1 \in S_1, e_2 \in S_2, s \in \text{state}\} \quad (2.10)$$

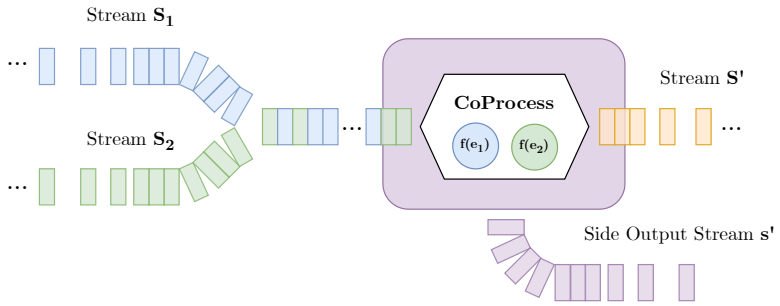


Figure 2.12: `CoProcess` transformation of Flink's *DataStream* API

The Flink *DataStream* API, in addition to the aforementioned sets of stream transformations, provides a comprehensive collection of windowing operators that enable fine-grained control over temporal data processing. Within the scope of this thesis we won't be leveraging window operators, while further information about them, may be found in Flink's original documentation [7]. Conclusively, the entire set of operators, combined with the rich capabilities, form a robust toolkit for stream processing, positioning Flink as a state-of-the-art component in modern stream processing use cases.

2.3 Apache Kafka

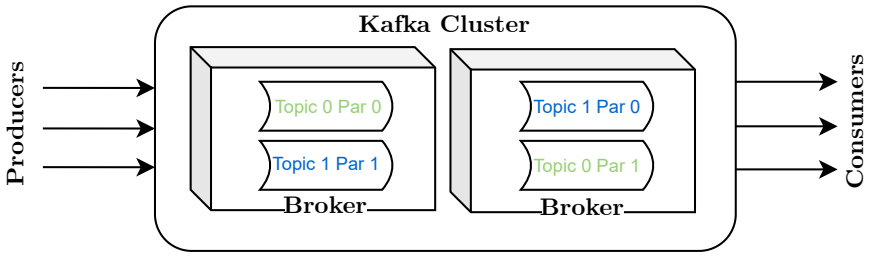


Figure 2.13: *Apache Kafka Cluster* architecture

Apache Kafka [8] is a state-of-the-art messaging broker, in terms of features and performance. A *Kafka Cluster* apart from a quorum of *Kafka Brokers* that are executed in parallel. Messages are organized in logical channels, termed as *Topics* and are represented as K-V pairs, generated by producers and ingested by consumers. *Kafka* splits *Topics* into *Partitions* which are distributed across the brokers ensuring parallelism. Each *Partition* maintains an ordered sequence of messages, assigning offsets and timestamp relevant to their ingestion order. *Kafka* outperforms, by far, other messaging brokers available today [9] positioning its adoption as an industry standard.

2.4 Data Summarization

While data streams serve as a very useful information form, they come along with a number of bold points that need to be addressed prior to their utilization. Streams potentially have an infinite nature. This fact, raises significant challenges in terms of storage and processing capabilities, as it becomes impractical to store or analyze every individual data point. Moreover, the volatility of stream information, where data can rapidly change or become outdated, necessitates real-time efficient processing to extract timely insights. Challenges like this are not limited to stream related processing. Even in the more simple case of dealing with plain large datasets, ranging from gigabytes to petabytes in size, handling the entire volume can be computationally and time expensive and sometimes even infeasible. To tackle these challenges and enable efficient processing of such massive and fast-changing data, innovative techniques like data sketches, which lie in the field of data summarization, have emerged as a solution.

Even though, this Thesis approaches the topic of persistency of synopses from an operational scope, a strong understanding of the fundamentals of data summarization & data sketches is considered essential. In this section we delve into the key points of data summaries, hereinafter referred to as **"Synopses"**.

2.4.1 Data Synopses

Data Synopses are compact, probabilistic data structures that provide approximate answers to queries over large datasets and unbounded streams. These structures maintain high accuracy in query answering while keeping error probabilities low. They are particularly advantageous because they minimize both storage and computational resources, all while preserving a useful approximation of the original dataset or data stream. They can be used as compact summaries of the original data, minimizing data processing and transfer costs when dealing with very large data volumes, thus offering scalability and energy efficiency. As noted by Garofalakis et al. (2012) [10], a synopsis of a large dataset captures essential characteristics of the original data while occupying significantly less space.

Synopses Properties

Considering all the aforementioned points, four highly favorable properties emerge for the nature of the synopses:

- One Pass Instantiation. They can be created by having access to the original data only once at arrival, usually the access is order-irrelevant.
- Memory & Time Efficiency. They require only a few amounts of physical memory to run on and can be updated in extreme rates.
- Composability & Portability. Due to their keyed and probabilistic nature they can be composed and queried in a distributed way. Using software engineering concepts they can be made portable and accessible to third-party systems, ensuring consistency.
- Accuracy Guarantees. Mathematical guarantees and proofs assure low error bounds, known a-priori, on every estimation retrieved from a synopsis.

Synopses Interface

From a technical point of view a synopsis may offer an interface for interacting with it, in terms of querying and updating it. The fundamental and minimal prerequisites a synopsis interface should expand to are:

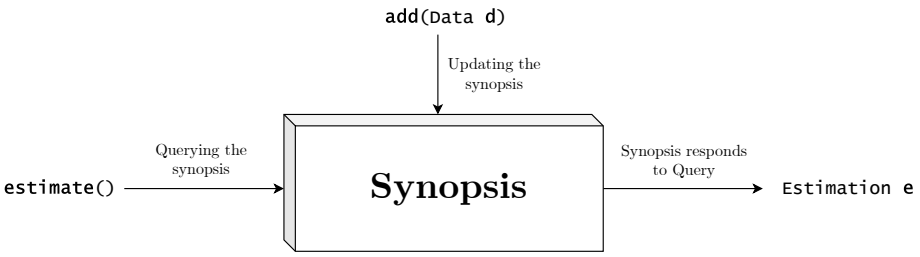


Figure 2.14: Synopsis from an implementation scope

- `add(Data d)`: Method for updating the current state of the synopsis by providing a new point of data
- `estimate()`→ `Estimation e`: Method for querying the synopsis for an estimation

The flexibility of a synopsis interface lies in its ability to support both basic and advanced operations. While `add()` and `estimate()` form the backbone of the interface, optional methods like `merge(Synopsis s)` and `delete(Data d)` enable it to address more sophisticated requirements and fit well in federated or distributed environments.

Synopses Families

Synopses vary per type, depending on the specific application and the nature of processing applied to the original data. The various summarization approaches outline 4 major families or synopses [10]:

- Sampling. Techniques like random sampling or stratified sampling select a representative subset (obtained stochastically) of the data to approximate the whole dataset.
- Histograms. They summarize the data distribution by partitioning the data into bins and storing the count of data points in each bin.
- Wavelets. A multi-resolution analysis of the data is provided, capturing both frequency and location information (image and signal processing).
- Sketches. Probabilistic data structures like Alon, Matias and Szegedy Sketch (AMS), Count-Min Sketch, HyperLogLog, and Bloom Filters, which provide efficient approximations for various queries, best performing on streaming data.

Each type of synopsis has its own strengths and weaknesses, and the choice of which to use depends on the specific requirements of the application, such as the type of queries to be answered, the acceptable error bounds, and the available computational resources and even the topology of the compute nodes.

2.4.2 Data Sketches

Data sketches have gained significant attention due to their versatility and efficiency in handling large-scale data. They are designed to provide quick, approximate answers to various types of queries running on streaming data, such as frequency estimation, cardinality estimation, and quantile computation, among others. Latest research efforts, highlight that state-of-the-art sketches may be outperforming traditional analytics methods.

Applications leveraging *Data Sketches* range from naive *single-attribute*[2] stream-processing to spatio-temporal and multi-variate dynamic analytics. Below we outline the state-of-the-art and other sketches that are often leveraged in modern research pursuits.

OmniSketch

Punter et al. in 2023 [1] introduced *OmniSketch*, a novel sketch allowing frequency query [3] answering on multiple attributes. An estimation can be provided on the fly, even if the set of attributes included in the estimation request are defined arbitrarily or are a subset of the initially declared ones at the time the sketch was instantiated. Unlike other methods [11, 12], *OmniSketch* does not require to know a-priori the total of subsets of attributes. A traditional approach would maintain a separate sketch for each subset of attributes, which would be computationally expensive and memory inefficient. *OmniSketch* is designed to avoid this, by using a hashing technique that allows the sketch to be queried for any subset of attributes, without the need to maintain separate sketches from the very beginning for each subset marking it a state-of-the-art both computationally and memory-wise.

In Figure 2.15 lies an example of a stream of data points representing a network traffic model with each tuple having 4 attributes: *source IP*, *destination IP*, *total length* and *DSCP priority*. Each message ingested, can also hold a 5th attribute denoted as *rid*

ipSrc	ipDest	totalLen	dscp
131.1.2.1	23.11.1.2	40	0
147.3.4.7	147.3.4.8	48	32
147.3.4.7	147.3.4.8	56	32
147.3.4.7	147.3.4.9	56	8
147.3.4.8	147.3.4.7	40	32
...	...	...	...

Figure 2.15: Example of a stream of data points ingested by *OmniSketch*

which is a unique identifier for the message. If not provided such unique *rids* can be easily constructed at ingestion time (e.g., using an arrival counter). The set of attributes of the streaming data can be expressed as:

$$\mathcal{A} = \{a_1, a_2, ..., a_{|\mathcal{A}|}\}, \quad \text{where} \quad |\mathcal{A}| = 4 \quad \& \quad a_0 = rid$$

[2]Querying a sketch for an estimation based on a single attribute of the ingested data

[3]Frequency querying in data sketches refers to the process of estimating the number of occurrences of an element in a large dataset, usually upon predicates.

The data structure of the sketch is represented in Figure 2.16. *OmniSketch* maintains $|\mathcal{A}|$ attribute sketches, of $d \times w$ dimensions, one per attribute similar to *Count-Min*. In each cell of the sketch, the min-wise [4] sample method described by Pagh et al. in 2014 [13] is maintained. The sketch is constructed in 3 three subsequent steps:

1. $\mathbb{S}0$:
   - The basic structure is an extension of the *Count-Min* sketch.
   - Each cell in $\mathbb{S}_0$ is a min-wise sample of the *rids* of the tuples that have been inserted into the sketch.
   - This step allows for efficient predicate-based frequency estimation.
2. $\mathbb{S}_0 \cap$ (The Estimator) :
   - Instead of taking the minimum across all rows (as in *Count-Min*) the estimator intersects all rows to reduce overestimation.
   - This fact leads to lower error bounds.
3. $\mathbb{S}1$ (OmniSketch) :
   - To achieve sublinear space complexity, K-minwise sampling is used.
   - Only a sample of *rids* is maintained in each cell.
   - Estimation is achieved by scaling up the sample intersection to approximate full count.

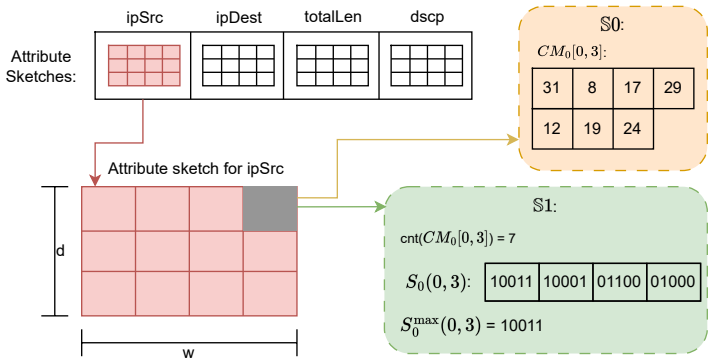


Figure 2.16: The *OmniSketch* Data Structure, The yellow-shaded box illustrates the contents of a cell in S0. The green-shaded box corresponds to the contents in S1

Each tuple of the stream is inserted to each of the $|\mathcal{A}|$ attribute sketches. Let us consider the example where an arriving tuple is inserted into the attribute search of *source IP*. A set of d hash functions of the form:

$$h_j(r_{ipSrc} \rightarrow [1...w])$$

[4] A minwise sample is a subset of elements from a dataset, selected by hashing all elements and retaining only the ones with the smallest hash values, enabling efficient estimation of set intersections and cardinalities.

are executed to discover the corresponding cells at each row j. When a corresponding cell is found, the *rid* of the tuple is inserted into the min-wise sample of the cell. Estimations are produced by hashing the desired predicate values, using the same hash functions, and intersecting the samples of the corresponding cells. The intersection is then scaled up to approximate the full count of the predicate.

CountMin Sketch

The Count-Min Sketch [11] is constructed as two-dimensional array of $w \times d$ dimensions. Primarily it is used for frequency estimation of items in a stream. Below we outline the key points of constructing, updating and querying a *Count-Min* sketch.

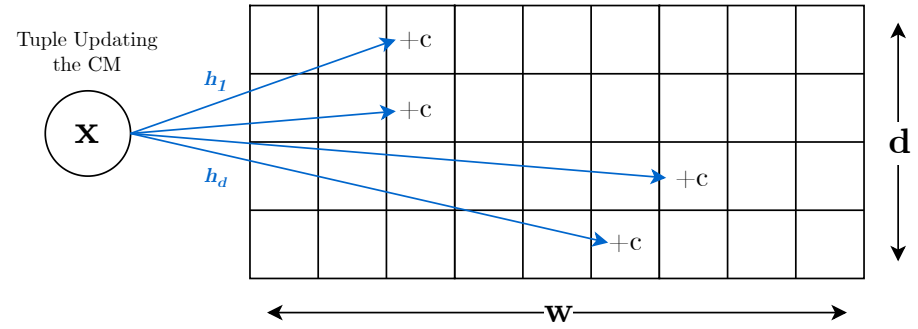


Figure 2.17: The *CountMin* Sketch Data Structure during an update operation.

- We maintain a $w \times d$ table of counters, initialized to zero.
- Let w and d control the accuracy and confidence of the sketch. Typically, $w = \lceil e/\varepsilon \rceil$ and $d = \lceil \ln(1/\delta) \rceil$ for desired error ε and failure probability δ.
- Each entry in the array is a counter, initialized to zero.
- We allocate d hash functions

$$h_j : \{\text{items}\} \to \{1, \ldots, w\}$$

  one per row, where $j \in \{1, \ldots, d\}$, $\quad |\text{items}| = n$.

In Figure 2.17 an update of a *CM* is shown. When an item x arrives in the stream updating the sketch by a quantity of c, it is hashed to row j and column $h_j(x)$ for every hash function h_d, so that one counter in each row is incremented:

$$\text{For } j = 1, \ldots, d, \quad \text{CMS}[j, h_j(x)] \leftarrow \text{CMS}[j, h_j(x)] + c$$

Frequency estimation is performed by querying the sketch for the minimum value of To estimate the frequency of an item x, the sketch returns the minimum counter across all rows:

$$\hat{f}(x) \;=\; \min_{j=1,\ldots,d} \text{CMS}[j, h_j(x)].$$

AMS Sketch

The AMS (Alon-Matias-Szegedy) sketch is a randomized algorithm for approximating frequency moments of a data stream using sublinear space. It is particularly effective in estimating F_2, which represents the sum of squared frequencies of elements in a stream.

The AMS sketch consists of a set of k hash-based counters, each computed using pairwise independent hash functions. The construction is as follows:

- Choose a set of k pairwise-independent hash functions $h_i : [n] \to \{-1, 1\}$ for $i = 1, \ldots, k$.
- Initialize k counters $Z_i = 0$ for all i.

When an element a_t arrives in the stream, the counters are updated as follows:

- For each $i = 1, \ldots, k$:

$$Z_i \leftarrow Z_i + h_i(a_t). \tag{2.11}$$

To estimate the second moment $F_2 = \sum_{i=1}^{n} m_i^2$, where m_i is the count of element i, we use:

- Compute $X_i = Z_i^2$ for each i.
- The final estimate is given by:

$$\hat{F}_2 = \text{median}\left(\frac{1}{k}\sum_{i=1}^{k} X_i\right). \tag{2.12}$$

Bloom Filter

Bloom Filters [14] are used to represent a set of n elements derived from a universe U. The Filter is a bit array of length m initialized to zero. The structure accounts for space-efficiency. It uses h_i hash functions to map elements to the bit array. When an element is inserted, the hash functions are applied to the element and the corresponding bits in the array are set to one. To check if an element is in the set, the hash functions are applied to the element and the bits are set. If all bits are equal to one, the element is estimated to be present in the set. Bloom Filters have a false positive rate, the occurrence of which can be controlled by the number of hash functions and the size of the bit array.

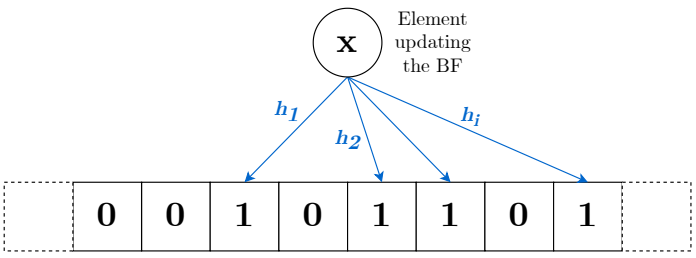


Figure 2.18: A *Bloom Filter* data structure during an update.

2.5 Stream Summarization Platforms

Within research and industry environments there is an ever growing need for real-time analytics on streaming data. As mentioned earlier, *Synopses* are competent enough to provide valuable and rich insights on data, with guarantees for low error boundaries. In the past years, numerous attempts have been conducted, aiming for a state-of-the-art solution combining both the power of *Synopses* and the ubiquity of *streaming analytics*. In this section, we will survey stream summarization libraries and platforms that have successfully integrated Synopses into streaming analytics and outline their key features and trade-offs.

2.5.1 Summarization SDKs

The foundation for leveraging Synopses in streaming analytics is an actually robust and modular software development library serving as a backbone for the integration. Apache DataSketches [15] is a well-known library that provides a wide collection of *Synopses* structures and algorithms, mainly suited for big data analytics. It covers a wide range of summarization algorithms such as *Count-Min Sketch*, *HyperLogLog*, and many others. *Stream-Lib* [16] is another Java library that provides a collection of *Synopses* algorithms, mainly focusing on streaming data. Cardinality estimation, frequency queries and filters are available within the artifact. Last but not least, *Streaminer* [17] is a collection of algorithms for mining data streams, including frequent itemsets, quantiles, sampling, moving averages, set membership and cardinality. It is implemented in Java and is available as an open-source project.

2.5.2 Condor

Condor [2] introduces a framework where synopses are treated as first-class citizens in stream processing, a significant improvement over existing platforms like Flink and Spark where summaries are not natively supported. Synopses are abstracted, they are modelled as stateful windowed aggregate functions. The framework's Architecture branches out into three main components: **Condor API**, **Condor Core**, and the external **Dataflow Engine** on which the framework is run. Each component is responsible for a specific task, such as managing and estimating the state of synopses, providing a collection of synopses algorithms, handling of ingesting and digesting data while also exposing a high-level API for users to interact with the system.

Interaction with the model is achieved through the *Condor API*, which provides a comprehensive way of designing any desired workflow. A workflow for *Condor* is dyadic, consisting of two main parts: *Processing* & *Querying*. Both parts are configured by the user by providing a set of parameters:

$$S = \{i, s, s', f_p, w, q, f_e\}$$

where i is the input stream, s is the type of synopses, s' is the synopses type of the output, f_p is a partitioning function defining a way to partition the input stream, w is the window type, q is the query stream and f_e is the evaluation function. The step of providing the parameters is denoted as **Step 1** in Figure 2.19

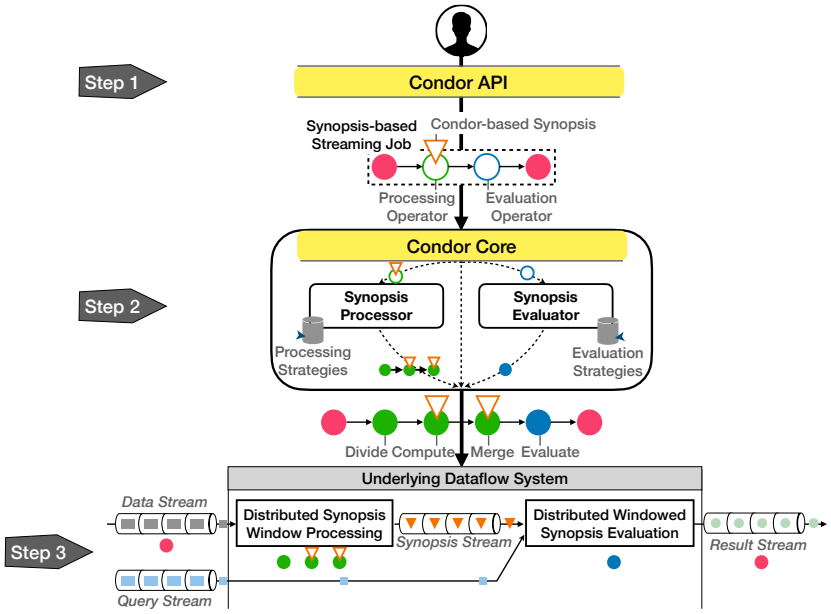


Figure 2.19: *Condor* [2] Synopses Streaming Framework Architecture.

Given the job's outline, the configuration is passed to the *Condor Core* which, based on the parameters, creates a new pipeline by leveraging the underlying *Dataflow Engine* as denoted by **Step 2** in Figure 2.19. The novelty in the pipeline execution lies within *Condor's* ability to elegantly select the optimal computing strategy for each of the 3 phases:

1. **Divide Phase**: Input data is evenly allocated to worker nodes.
2. **Compute Phase**: Parallel partitioned synopses are maintained across workers and updated.
3. **Merge Phase**: Partitioned synopses are merged and the final result is computed.

The results produced during the *Merge Phase*, are propagated to the *Evaluator* which provides the final output to the user (**Step 3**, Figure 2.19) by optimizing the strategy on extracting it. Conclusively, as a prominent work *Condor* introduces various novelties, filling a gap in the summarization analytics. Trade-offs are not lacking though. Horizontal scalability is a concern and not guaranteed by the the architecture, while federated and vertical scalability are not supported at all.

2.5.3 StreamApprox

Do le Quoc et al. [18] introduced *StreamApprox*, serving the purpose of a stream analytics system implemented on top of an algorithm. *StreamApprox* presents initially a *Reservoir Sampling Algorithm* used to estimate queries and produce approximate output. In contrast with other efforts made in the past, the work's novelty lies in the fact that the system is capable of answering queries on both streaming and batch data. The *Reservoir Sampling Algorithm* was implemented as a prototype operator on *Apache Flink* and *Apache Spark*. Provenly, *StreamApprox* is capable of providing rapid results and even run in a distributed deployment. In short *StreamApprox* is a promising work, but it only supports sampling while it falls behind in scalability options.

2.5.4 STREAM

STREAM [19] (STanford stREam datA Management) is a prominent but groundbreaking work for its time, envisioning an integrated system for answering continuous queries [5] over data streams. *STREAM* was introduced as a survey on a *DSMS* with the vision to highlight, later on, the need for real-time analytics. The proposed architecture for processing queries and answering them continuously is depicted abstractly in Figure 2.20.

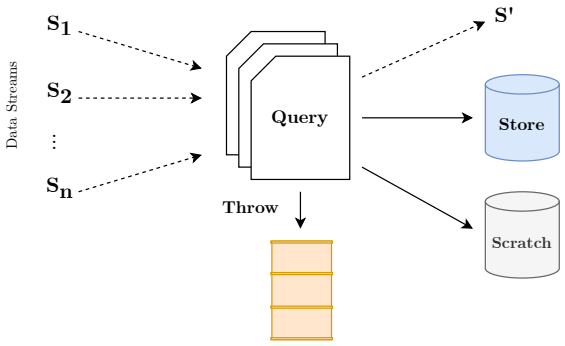


Figure 2.20: *STREAM* [19] Proposed Architecture.

Total of n streams are ingested in the architecture and their data points undergo through a set of running queries of some function f_q^i. Let us consider a query $\mathcal{Q}$ with an answer $\mathcal{A}$. Let t be each new tuple ingested through the relevant streams. $\mathcal{Q}$ is answered in a number of non mutually-exclusive conditions:

1. $\mathcal{Q}$ can pass-through tuple t to S' if it can tell apart it is a new tuple and it must be destined to $\mathcal{A}$.
2. If t is a new tuple that should be in $\mathcal{A}$, but may be some time no longer be in $\mathcal{A}$ then is directed to *Store*. This way, S' and *Store* define the current $\mathcal{A}$.
3. A new tuple t may cause tuples to be moved from *Store* to S'.
4. t's data may be needed to be stored for future answers of $\mathcal{Q}$. In this case, t is directed to *Scratch*.
5. If t is not destined to answer any more $\mathcal{Q}$, then it is directed to *Throw*.
6. It is not uncommon that tuples originating to S' from *Scratch* may be destined to *Throw*.

STREAM's architecture might seem trivial in comparison with modern, state-of-the-art frameworks and applications. Despite that, the need for stateful stream processing and query answering emerges through it. Real time answers are considered a key-feature of streaming analytics, but recallability on historic data or on previous states is also considered a necessity, although not being a straight-forward accomplishment.

[5] Receiving constantly and in real-time an estimation of a function f_q upon or not on data arrival, over a stream or stored data.

2.6 Synopses Data Engine

Synopses Data Engine (SDEaaS) was designed and developed by Kontaxakis et al. [6, 3, 20] as a fully integrated data processing engine accommodating *Synopses* as key-components for on-demand data summarization on ingested streams. It was introduced as a *SaaS*, able to interconnect with platforms and applications, allowing for analytics and data summarization to be provided in the form of a service. The engine is built on top of *Apache Flink*, exploiting the *DataStream API* to implement a scalable and efficient *Dataflow*. *SDE*'s novelty, stands on its ability to handle requests for on-the-fly maintenance and querying of *Synopses* on ingested data streams. This aspect of it, marks this work as the state-of-the-art and a unique effort its field, that provides an end-to-end solution with minimal trade-offs. The request-oriented nature of *SDE*, positions *Synopses* themselves as an on-demand service (*SDEaaS*). In this section, we will provide an in-depth analysis of the architecture, model, implementation, and key features of the engine. Later on, we will assess *SDE* as of its performance, scalability and extensibility. Consequently, this section, lays the groundwork, enabling us to champion our vision on making *SDE* an integrated and stateful component of a Knowledge Ecosystem.

2.6.1 Architecture

The core components of *SDE*'s architecture are *Apache Flink* [4] and *Apache Kafka* [8]. *Flink* is used as the main processing engine, while *Kafka* is used as the main message ingestion and distribution system. The architecture of *SDE* is depicted in Figure 2.21. The engine, as we will exhibit in the following sections, is built as a *Dataflow* model ready to be run in any Flink Cluster of any size. The *Dataflow* is composed in Java and then packaged as a *JAR* file to be submitted as a *Flink Job* for execution. The *Flink* Cluster handles the execution of the submitted job and ensures its consistency. The *SDE* job transacts with *Kafka* through the respective topics to ingest data and requests and digest estimations and results. Users, applications and third-party entities may interact with *SDE* through the provided *Streaming API* to submit requests, stream data and retrieve results. The *SDE* is designed to be scalable and multiple instances of the *Dataflow* can be run on the same or different *Flink* Clusters, which can possibly be geo-dispersed, allowing for parallelization and federation of the compute process. Communication happens through a common *Kafka Cluster* on the respective topic, used to establish synchronization and consistency among them.

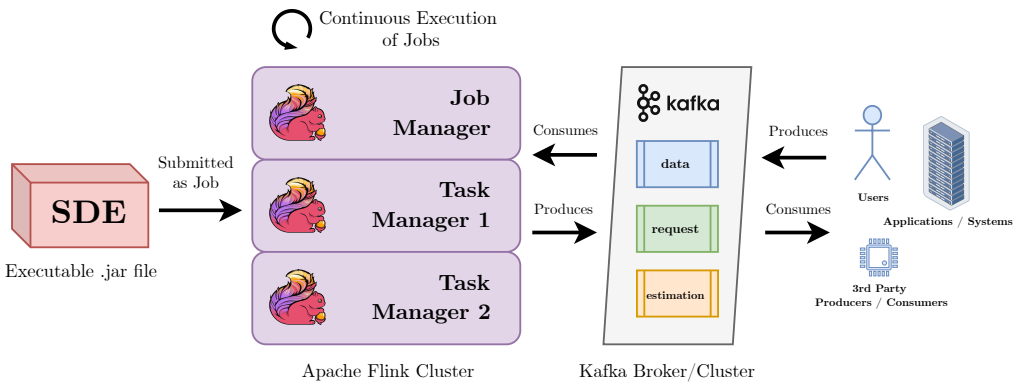


Figure 2.21: *Synopses Data Engine* (SDEaaS) Architecture

2.6.2 Parallelization Schemes & Ingestion Modes

SDE supports a variety of *Synopses* that can be maintained, updated and queried at any given time. Ingested data, destined to update an already maintained *Synopsis(es)*, may find their way to it based on a key. This key can be the `StreamID`, in the tuple belongs to an unbounded stream or `DatasetKey` in case the tuple belongs to a streamed static dataset. This, defines 2 ingestion modes supported by the *SDE*:

- **Stream Ingestion**: Each tuple is assigned a `StreamID` which is later used to route the tuple to the appropriate set of *Synopses*.
- **Dataset Ingestion**: Each tuple is pre-assigned a `DatasetKey`, which is later partitioned and used to route the tuple to the appropriate set of *Synopses*.

SDE takes advantage of *Flink*'s parallel computation capabilities in a sophisticated way ensuring integrity in the processing steps. The parallelization scheme employed in the *Synopses Data Engine* is based on Key-Partitioning. This suggests, that each ingested tuple carrying a specific key and is intended to be included in one or more maintained *Synopses* does so based on the key it is assigned to it. Two types of Keys are supported:

- **Stream Based Keys**: Ensures all tuples originating from a stream are routed to the same parallel worker.
- **Dataset Based Keys**: Ensures all tuples originating from a dataset are routed to the same parallel worker by assigning a partitioned key to the dataset tuples as they are ingested.

2.6.3 SDE Synopsis

As mentioned earlier, a *Synopsis* provides an interface related to the available operations that may be performed on it. *SDE* follows this approach to implement a variety of *Synopses* in Java, offering a rich set of methods that can be applied per type and per case. *Synopses* are modelled through an abstract class `Synopsis`, illustrated in Figure, that is inheritable by any desired Java Class modelling a certain type of *Synopsis* (AMS, OmniSketch. HLL, BloomFilter, CM etc.).

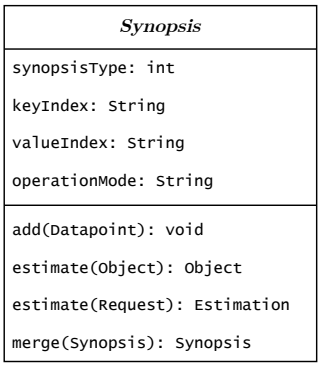


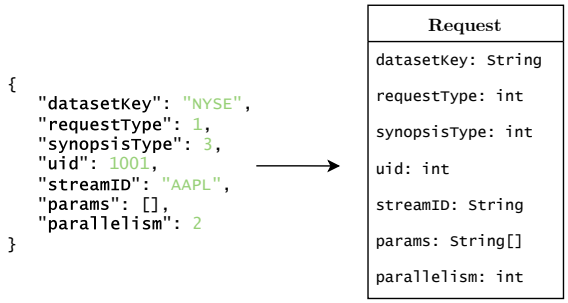
Figure 2.22: SDE *Synopsis* Java Class

2.6.4 Data, Requests & Estimations

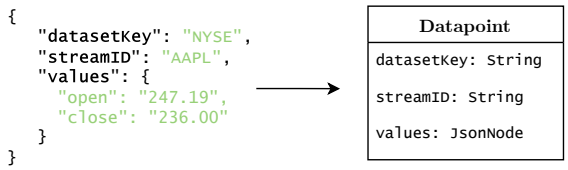
As mentioned earlier, *SDE* ingests data and requests and emits results by interacting with a *Kafka Cluster* through appropriately set up, topics. A total of minimum 3 topics are required for the engine to operate. The `data` topic is used to ingest the stream of inflowing data, the `request` topic is used to ingest the stream of incoming requests, and the `estimation` topic is used to emit estimations computed within the engine's context. Topic names are not strictly set and can be configured prior to the execution of the *Flink Job*.

In order for all parts of the architecture to be on the same page, ingested and digested tuples are transacted in the form of *JSON* objects and within the engine's *Dataflow* context, they are parsed and modeled as *Java Objects*. Below we provide a brief overview of the structure of each type of object that will help us address major parts of this Thesis in the following chapters:

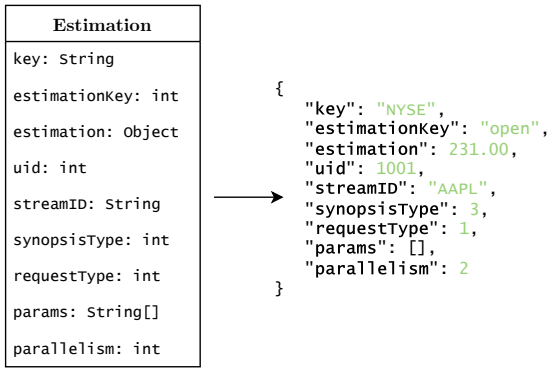
- `Request`: A model representing any ingested request. Each JSON object is parsed and an object of the class is instantiated.



- `Datapoint`: A model representing any ingested data point. Each JSON object is parsed and an object of the class is instantiated.



- `Estimation`: A model representing any emitted estimation. Each object is converted to JSON and emitted.



2.6.5 Streaming API

As *SDE* is intended to operate as a service, it provides a *Streaming API* with rich capabilities to interact with the engine. The interaction with the API is achieved through streaming requests sent as messages to the respective *Kafka Topic* from which the engine consumes them. Requests ingested by the engine mainly relate to manipulating the state of *Synopses* and retrieving estimations. The following types of requests are currently being handled:

- **Add Synopsis**: Maintain a new *Synopsis* in the engine. Expands to :
  - **Add Continuous Synopsis**: Maintain a *Synopsis* that is continuously queried after each update.
  - **Add Randomly Partitioned Synopsis**: Maintain a *Synopsis* of which the state is randomly partitioned.
- **Delete Synopsis**: Remove a *Synopsis* from the engine.
- **Estimate Synopsis**: Query a maintained *Synopsis* for an estimation.

Previously we illustrated the *Request* class that models the API's requests. Here we outline the fields of the request class:

- `datasetKey`: The key related to a *Synopsis* maintained for a specific dataset.
- `streamID`: The key related to a *Synopsis* maintained for a specific set of streams.
- `requestType`: The type of the request to be handled (`add, delete, etc.`).
- `synopsisType`: The type of the *Synopsis* to be maintained (CM, BloomFilter, etc.).
- `uid`: The unique identifier of the request.
- `params[]`: The array containing special parameters.
- `parallelism`: The degree of parallelization that should be applied in the action of the request.

When a request, formatted in JSON, is propagated through a *Kafka Topic* as a message, encoded in bytes. On its path outside the engine a request has the following form:

```
Value <key, streamID, requestType, synopsisType, uid, params, parallelism>
```

A detailed overview of the requests supported by the *Streaming API* of the *Synopses Data Engine* may be found in the Appendix A of this Thesis.

2.6.6 Dataflow Model

In order to get a stronger understanding of how the *SDE* operates in its core, we will perform a detailed overview of the *Flink* Dataflow Model that is designed to implement the actual engine. The *Dataflow* model is depicted in Figure 2.22. From a high-level perspective, the *Dataflow* handles 3 external streams either as input or output.

The **input** streams are:

- **Data Stream**: The stream ingested from the `data` topic that carries the data to be destined for summarization.
- **Request Stream**: The stream ingested from the `request` topic that carries the requests to be handled.

The **output** streams are:

- **Estimation Stream**: The stream producing the estimations retrieved upon request from the core of the engine to the `estimation` topic.

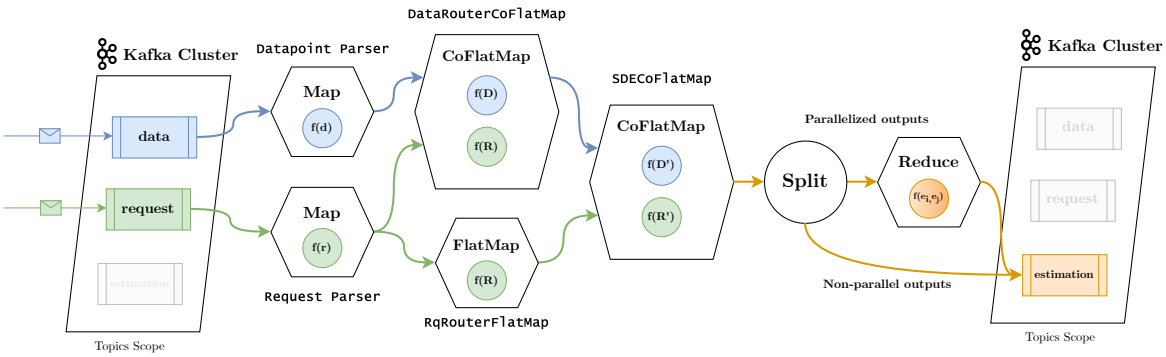


Figure 2.23: *Synopsis Data Engine* (SDE) *Dataflow* model

Let the green coloured path be the path that requests follow on their way through the engine and the blue coloured one, the path data follows. As illustrated in Figure 2.23 the *Dataflow* consists of a chain of *Flink* operators propagating the ingested tuples through them.

Parser Operators

When a data tuple or request arrives through the corresponding topic and is consumed by the engine's first layer it goes through an appropriate *Parser*. These two components are denoted as `Datapoint` and `Request` parsers in our illustration. Their role lies on converting received messages, formed as JSON objects, to either `Request` or `Datapoint` objects that are going to make their way through the engine carrying the initially raw information in a structured form.

Router Operators

`DataRouterCoFlatMap` and `RqRouterFlatMap` are classic `FlatMap`, `CoFlatMap` operators, of *Flink's DataStream API* that handle and apply the parallelization schemes *SDE* supports.

- `RqRouterFlatMap`: Handles the assignment of keys to *Requests* such that they reach the appropriate parallel worker(s).
- `DataRouterCoFlatMap`: Keeps track of already propagated requests that intended to maintain new synopsis. Ingests both *Datapoints* and *Requests* and assigns keys to *Datapoints* according to a maintenance requests that has already been streamed through the operator and their action was registered.

The role of both of the *Routers* is considered crucial for the integrity of the engine in terms of guarantees on a robust parallelization scheme.

Maintenance Operator

`SDECoFlatMap` is the core of the engine, as this is where *Synopses* reside. This operator runs on every parallel worker that is deployed upon to the configuration of the *Flink Job*. It handles both *Requests* and *Datapoints* to either manipulate the state of the maintained *Synopses* upon a desired action or update them using the value carried by a *Datapoint*. Every tuple destined to this operator has been previously been keyed accordingly, in order to reach every parallel instance of it, running across each worker. This *CoFlatMap* operator is the main point that may be leveraged to interconnect and expand the engine's capabilities as the entire information about each *Synopsis* is maintained there. Estimations are emitted through it and get directed to the `Split` function later on.

Split Function

Estimations that are, for some reason, produced with unitary parallelism are emitted directly to `estimation` topic of the *Kafka Cluster*, while tuples originating from parallel computations are directed to the `Reduce` operator to aggregate the parallel answers into one using an appropriate *Reduce Function* that varies upon the type of the *Synopsis* from which the estimation was generated.

Reduce Operator

The `Reduce` operator aggregates the results of parallel computations aiming for an estimation into a single result. This is necessary when the estimations are produced by multiple parallel instances of the `SDECoFlatMap` operator. The `Reduce` operator applies a *Reduce Function* to combine the partial results into a final estimation. The specific *Reduce Function* used depends on the type of *Synopsis* being queried for that estimation undergoing processing. Once the reduction is complete, the final estimation is emitted to the `estimation` topic of the *Kafka Cluster*.

2.6.7 Overall View

SDE stands out as the most auspicious and sophisticated work, designed for on-demand data summarization. Its support for various ingestion modes and parallelization schemes, ensuring scalable data processing, the extensible *Dataflow* model, and streaming API facilitate seamless integration and performance. In the following chapters, we will explore the extension and configuration of *SDE*'s capabilities. Our aim focuses on making *Synopses*, being maintained within the *SDE*, portable and persistent with enabled versioning, towards making the engine an interlinked stream summarization tool.

2.7 Data & Knowledge Lakes

Data Lakes represent an architectural paradigm of software-engineering within the field of *Data Management*. Designed to ingest and store large volumes of heterogeneous data in its native, untransformed format, they are one of key-components, serving as the first layer of data ingestion in many business and research environments. While classic data warehouses, require a pre-defined schema before data can be stored, *Data Lakes* adopt a flexible schema-on-read approach. This methodology empowers the storage of raw, unstructured, semi-structured and structured data, thereby offering unprecedented flexibility for subsequent processing and analysis that may be performed on them.

The primary purpose of data lakes lies in serving as a centralized repository that allows data scientists, researchers and usually enterprises to explore and analyze datasets without the limitations imposed by predetermined data models. By preserving the original fidelity of the data, these repositories facilitate advanced analytics, machine learning, and research, which can extract latent insights, drive decision-making and direct strategies across various domains.

Historically, platforms like *Microsoft's Azure Data Lake* [21] have played an important role in demonstrating the practical benefits of the *Lake*. *ADL*, brings together services like Azure Data Lake Store and Azure Data Lake Analytics, integrating scalable cloud storage with powerful analytics capabilities, thereby laying the groundwork for subsequent innovations in big data management. Similar efforts, including Hadoop-based frameworks and Amazon S3–backed data lakes, have further validated the efficacy of storing large, raw datasets to support a wide range of data-driven applications and processes.

Knowledge lakes, have their foundations on the *Data Lake* concept, while also incorporating semantic layers, analytics layers and metadata that transform raw data into structured, contextualized knowledge ready to be delivered to research or decision-making flows. Instead of entirely relying on schema-on-read concepts, they integrate tools such as natural language processing, machine learning, and graph-based representations to enrich the stored data and interpret them. This added layer of semantics allows for the creation of knowledge graphs and ontologies that capture relationships between initially raw data. The main layers structuring a *Knowledge Lake*, as illustrated in Figure 2.24, are:

- Data Ingestion Layer. Collection of data from diverse sources.
- Scalable Data Storage. Holds structured, semi-structured or unstructured-data.
- Metadata & Semantic Layer. Metadata registry for indexing.
- Processing & Enrichment Layer. Transforms raw data into actionable insights.
- Knowledge Graph Layer. Link data points and highlight relationships.
- Query & Search Interface. Enable search, interaction, and visualization.
- Security & Governance Layer. Ensure data quality, security, privacy, and compliance.

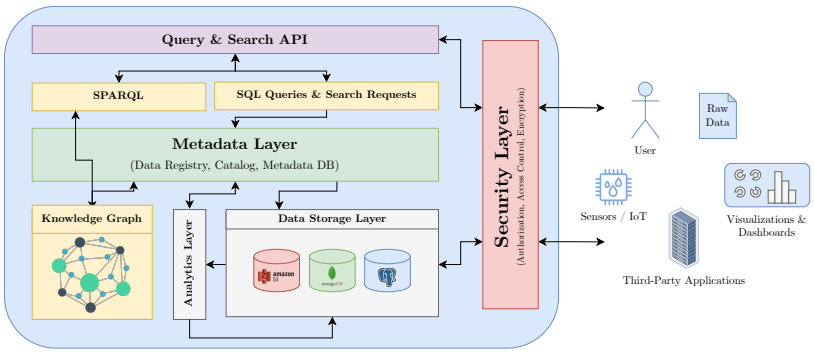


Figure 2.24: *Knowledge Lake* Example Architecture

2.7.1 Knowledge Generation in Lakes

Depending on the use-case the *Knowledge Lake* is intended for, various versatile data analytics and profiling tools are incorporated either as third-party producers or integrated components of it. Such tools, compose the analytics layer of a *Knowledge Lake*. Ranging from LLMs, profiling and summarization tools, they enable the extraction of meaningful insights and the generation of knowledge from raw data present in the *Lake's* data storage layer.

2.7.2 STELAR KLMS

The *STELAR (Spatio-TEmporal Linked data for the AgRi-food data space)* [5] is designing and developing a *Knowledge Lake Management System (KLMS)* that will be used to curate raw data lakes into *Knowledge Lakes* related to the the Agri-Food Data Space. *STELAR's* innovation stands in the combination of specialized analytic tools and a robust, secure and versatile platform used to house the data and the a great set of components and services. The vision of the project is encompassed in the following aims:

- Facilitating & Improving Data Discovery.
- Supporting Data Linking and Interoperability.
- Improving the annotations of the data and supporting synthetic, AI-ready data generation

At its core, the *STELAR KLMS* closely aligns with the reference lake architecture depicted in Figure 2.24. However, it also incorporates notable enhancements and innovations that distinguish it from the conventional models. The core is developed to be flexible and deployable in many platforms with plug-n-play scalability options.

As illustrated in Figure 4.1, *STELAR* employes a *MinIO Object Store* [22] as a data layer rather than having to choose from a range of relational and non-relational databases that impose limitations regarding their interoperability. This fact suggests, that every single of piece of data is treated as a file during its lifetime within the *Lake*. *CKAN*, a data-catalog serves as the metadata layer, registering information and relations between objects lying within the *MinIO* storage. At the same time, the catalog is extended to operate as a centralized repository for the data analysis and interlinking tools, registered

and ready to be executed, in a *STELAR Cluster* [6]. Security and authorization governance are enforced through *Keycloak* [23] by leveraging its flexibility on applying the *OpenID Protocol*, alongside a range of custom features. Every component establishes an *OIDC* connection with the *IDP* and an *SSO*[7] scheme bounds the cluster together.

STELAR stands out among traditional *Knowledge Lakes* by dint of its ability to offer an interface for incorporating multi-step workflows by leveraging the *KLMS Tools*. Workflows can optionally be orchestrated by an external workflow engine communicating with the Cluster using the *Python SDK*. A tool execution, either if it is run in the cluster itself or remotely, publishing its results through the SDK, is tracked by the *Data API* and metadata are recorded. Those metadata and results are the actually valuable insights that consist *Knowledge*. Among others, tools in the *KLMS Toolkit* expand from satellite imagery analyzers, dataset profilers and entity matching using LLMs to correlation detective tools and even a stream summarization framework, the *Synopses Data Engine*.

Generated *Knowledge* is exposed to users, engineers and data scientists via a rich *Data API*, backed up by a *Python SDK Client* and a user-friendly graphical interface, which all intercommunicate, on the one hand with the *Data Catalog (CKAN)* to extract actionable metadata, and on the other hand with the *OnTop Knowledge Graph* to outsource relationships and links between them.

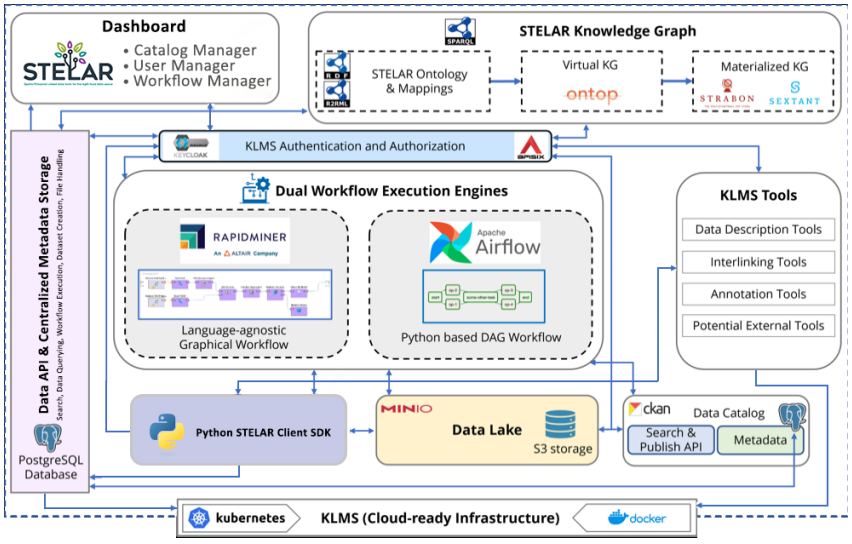


Figure 2.25: The Architecture of *STELAR KLMS*

[6] A deployment of the *STELAR KLMS* in an on-premise environment within an enterprise or a research institution.

[7] SSO (Single Sign-On) is an authentication process that allows users to access multiple applications within an organization or interconnected services, with a single set of login credentials,

2.7.3 SDE within STELAR KLMS

The *Synopses Data Engine (SDE)* was destined to be a core component of the *STELAR KLMS*, designed to generate and manage data synopses as part of processing workflows. Having discussed thoroughly the nature of *Data Synopses* in Section 2.4.1, it is well established that they are highly volatile as they mainly reside in memory. The same applies in the *SDE's* case, *Synopses* are maintained and updated as long as the *Flink Job* is up and running. Considering *STELAR's* data-centric vision, an important question that arises is how the insights generated during the runtime of the *SDE* will be able to be stored and updated in the future without eventually having to original never again.

In the context of *STELAR*, *Data Synopses* supported by the *SDE* are utilized in various scenarios and mainly under the hood (i.e., transparently to the user). Assessment of meteorological data in *Pilot 3: Timely precision farming intervention* and real-time analytics or even inclusion of generated estimations in the *Data Profiler's* extracted results are made possible through *SDE*. By leveraging synopses, *STELAR*, can significantly reduce the computational resources required for data processing, thereby enhancing performance and reducing its energy footprint.

Overall, the integration of the *Synopses Data Engine*, enriched with the new features for persistency and interoperability, within the *STELAR KLMS* exemplifies the innovative approach of the project in optimizing data management and analysis, ultimately contributing to more efficient and actionable knowledge generation that is persistent and highly available. In the next chapters of this Thesis we are going to break down the process of expanding *SDE's* capabilities and we will outline the way we addressed challenges opposed during the integration process in future work.

Chapter 3

Persistency in SDE

In this chapter, our focus shifts to a detailed overview of the design and implementation of persistency in the *Synopses Data Engine*. We will thoroughly outline the architectural considerations, software choices, and technical challenges, that facilitated the extensions made in the engine to support persistent, portable and reusable snapshots of *Synopses*, residing within it. This part accounts also, for both functional and non-functional requirements, ensuring an in-depth analysis of the mechanisms used to enable durable storage.

3.1 Preliminaries

Persistency, in software engineering, refers to the ability of a system or application to retain data beyond the runtime, preventing data loss even after shutdowns or failures. It is well established, that any modern software platform should be able to maintain state and data per use-case. This is achieved by storing data in non-volatile storage mediums such as databases, file systems, or cloud-based object stores, allowing it to remain retrievable and most of the time updatable.

Persistent Storage is distinguished from other storage schemes, by its ability to maintain a history on versions of store data. This mechanism allows for the construction of lineage between previous and future states data will go through, forming a directed acyclic graph (DAG) or a tree-like structure that traces the evolution of a data entity or object over time. The ability of a system to track states data have been through, holds an important role in ensuring integrity, rollback capabilities, disaster recovery and historical analysis, especially in data ecosystems where records are frequently getting modified.

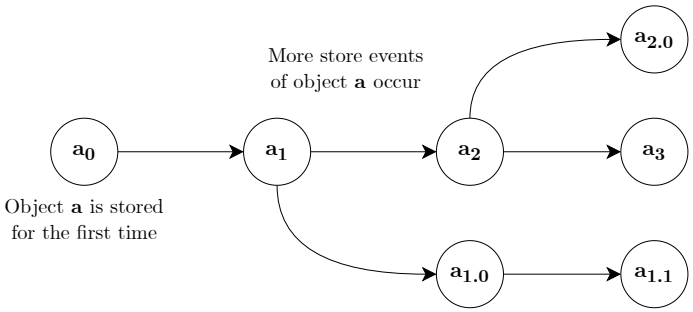


Figure 3.1: A *Persistent Storage* version tree of an object *a*

On the way of integrating *SDE* in the *STELAR KLMS*, versioning of *Synopses* was considered a key feature that needed to be incorporated. *STELAR* depends strongly on both real-time data and historic analytics produced upon past data, thus *Synopses*, with their highly volatile nature could not serve the purpose without a robust framework to support a snapshot mechanism, applied to them. *SDE* helped us address this challenge. With the main pipeline already implemented, we intervened and added the new required capabilities, which are going to be thoroughly discussed in the following sections.

3.2 Design & Requirements

Prior to demonstrating the design of persistency applied to the *SDE* we will outline the requirements and challenges upon which the foundations of the new features were laid. As long as the fundamentals have been well established, we will focus on analyzing the actual design for embedding persistency in the *Synopses Data Engine*.

3.2.1 Non-Functional Requirements

Synopses Data Engine was developed upon high-standards regarding its performance in terms of simultaneous stream handling and maintenance of numerous *Synopses* in the order of thousands, often in parallel. This fact, sets a standard for the efficacy and behavior of the engine itself, which should adhere to the original benchmarks. Below we quote crucial non-functional requirements that were guaranteed and should be met once our intervention is completed.

1. Performance. The ingestion of data should not be stalled or underperform while snapshots are taken. A component handling the snapshot must operate with guarantees that the process completes rapidly, and it does not interfere with the main flow of data or requests.

2. Parallelization. As noted in Section 2.6.2, *SDE* applies a custom, sophisticated parallelization scheme, that ensures each tuple of data or requests reaches the appropriate parallel worker maintaining a part of the *Synopsis* they are destined to. Since this is a core feature of the engine, and serves as the main concept for assuring that *SDE* is indeed horizontally scalable, it should not be abrupt. Later on, we will address on how the parallelization scheme must apply to *snapshots* as well, to ensure intact parallel computations.

On the other hand, new features while adhering to the aforementioned quality control points, should also provide guarantees for the following:

1. Portability. The generated *Synopses Snapshots* should be portable and transferable to third-party services or to other *SDE* deployments. This property also foresees the integration of the engine in *STELAR* where *Snapshot of Synopses* will be treated as data units/objects.

2. Usability. The *Snapshots* are subject to be used to be readable when loaded into the engine on-demand or on disaster recovery scenarios.

3. Fidelity. The *Snapshots* captured, must represent the given *Synopsis* as is, without altering its format or state at the given time. When loaded back to the engine the *Synopsis* acts seamlessly as if it was running forever in it.

4. Persistency: Snapshots occurring from the same *Synopsis*, should be linearly connected formulating a DAG (Directed Acyclic Graph), preserve succession and so enable versioning.

5. Parallelization: Snapshots occurring from a *Synopsis* being maintained in parallel workers, should also be acquired in parallel. This way, when loaded back to the engine, the state of the *Synopsis* is resumed identically.

3.2.2 Functional Requirements

Since the foundational behavioral properties of our design have been set, we will delve into the actual features that are expected in the context of *Synopses Snapshot*. These features will expand the system's capabilities by enabling efficient state preservation, retrieval, and instantiation of *Synopses*.

On Demand Snapshot of a Synopsis

Among others, the most fundamental feature that needs to be added to the engine, is the ability to capture a *Snapshot* of a running *Synopsis* and store it in *MinIO* as an object. The capturing should be able to take place multiple times, such that a lineage could be achieved between past and future snapshots. Since the snapshots should be able to be captured on demand the *Streaming API* of *SDE* should be expanded appropriately to support it via a new request type.

Loading the latest Snapshot to a running Synopsis

One, may be offered the choice to overwrite the instance of a running *Synopsis*, with the latest existing *Snapshot* (if any) of it. This also sets a requirement for expanding the existing *Streaming API* to support it.

Loading a specific Snapshot to a running Synopsis

In addition to the previous desired feature, a specified *Snapshot* version, including those predating the most recent one, must be able to overwrite its corresponding running *Synopsis*. This feature ensures that historical states can be retrieved and reinstated.

Instantiate a new Synopsis based on an already existing Snapshot

An existing *Snapshot* should be able to be loaded back into the engine and based on it, a new *Synopsis* is instantiated. The newly added *Synopsis* should be identical to the one, in terms of state, for which the *Snapshot* has been previously captured.

All the aforementioned features, highlight that the existing *Streaming API* (2.6.5) requires to be extended to fully support the new functionality. In Section 3.3 we detail the additional request types that were added in the *SDE's framework*, among their specifications. The relevant documentation may be found in the Appendix A of this Thesis.

3.2.3 Storage Platform

When it comes to choosing a platform for data storage the variety of available options expands to cater to different needs that are subject to vary per use-case. The form of data, the required scalability options, deployment site and eventual privacy constraints can direct and limit the amount of matching solutions. An engineer, nowadays, may pick from a vast collection of storage services, such as relational databases, object stores, network file systems or solutions combining a subset of the aforementioned ones. Each of them, comes with its own strengths and weakness, establishing this way the need for thoughtful assessment of trade-offs imposed per case.

Early on, in our survey, it seemed that neither a classic relational database model would fit the case to serve as the storage layer for *SDE*, nor a document-based (non-relational) one. We moved forward, by honoring the advanced and versatile features, platforms based on the *Amazon S3* protocol have to offer. Our selection was not biased or inconsiderate, instead it got driven by three major factors:

1. Synopses Format: *Synopses* supported by the *SDE* are many and usually differ from one another regarding their structure. This fact, alongside our aim to maintain the flexible nature of *SDE* doomed the idea of schema-based solutions, where for each *Synopsis* a potential table-schema would be needed, limiting this way support for new *Synopses* to be adapted.

2. Compatibility & Flexibility: Services built on *S3* connect to a broad set of third-party applications and does not require any particular schema applied to the stored data.

3. Stelar Data Layer: *S3* is used within the *STELAR KLMS*, serving as the data layer of the *Knowledge Lake*. This would allow for seamless integration of *SDE* as a part of it.

S3 (Simple Storage Service) [24] was launched in 2006 as a scalable, durable, and highly available object storage service for developers and enterprises. It was initially released and is currently maintained by *Amazon Web Services*. Since then, *S3* has become an industry standard when it comes to storing big or small objects in the cloud. As of today, 400 trillion objects [25] are stored within *Amazon S3* and this number increases constantly. Our intent on not limiting *SDE* to a single storage-provider, is well-served by our choice of *S3*. Many open source solutions for object storage have emerged in the latest years providing performance and scalability guarantees and simultaneously supporting fully the *S3* communication standard.

MinIO Object Store [22] is an open-source alternative to *Amazon's* monetized cloud. *MinIO* offers high-performance, vertical and horizontal scalability across nodes while also being a self-hosted solution, providing this way also strong privacy guarantees. The API of *MinIO* was being developed, since the very beginning, adhering to *S3* standards, eliminating this way the need for users, working with the official *Amazon S3*, to go through reconfiguration processes when switching to *MinIO*. This seamless compatibility ensures that applications and services built to work with *Amazon S3* can interact with *MinIO* without core modifications, making migration effortless. End users can leverage *MinIO's* custom-built SDKs, available in multiple programming languages, or connect with their instance using the official *Amazon S3 SDKs* for enhanced flexibility and ease of adoption.

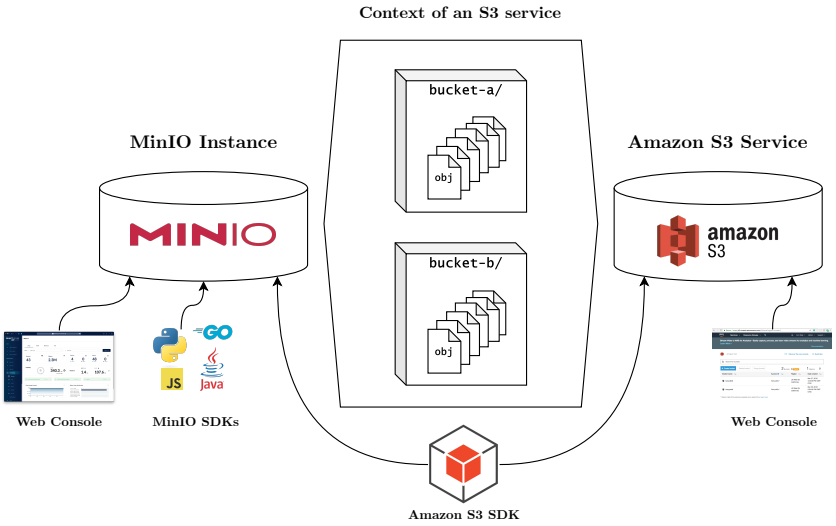


Figure 3.2: Architecture of *MinIO* & *Amazon S3* object storages

Data units in a *MinIO* storage are termed as *Objects*, which are ultimately files. *Objects*, as illustrated in Figure 3.2, are organized in buckets of any name. It is worth noting that *MinIO* maintains a path-like nomenclature scheme, meaning that every object can be named using a hierarchical structure similar to file system paths. This allows for intuitive organization and retrieval of objects stored within a bucket. For example, an object stored in *MinIO* could be named as:

```
bucket-a/documents/reports/2025/metrics.json
```

In this case, `bucket-a` is the bucket name, and `documents/reports/2025/metrics.json` represents the object's name and path within the bucket. Although *MinIO* does not inherently support a folder structure, this approach enables users to mimic directory-like arrangements for better structure in their data.

3.2.4 From Synopses to Snapshots and Back

One of the challenging parts of this work, lies in defining the way *Snapshots* of *Synopses* would be captured during the runtime of *SDE*. Additionally, a complementary non-trivial matter was how *Snapshots* would be loaded back in the engine to overwrite or instantiate *Synopsis* objects. In *SDE* the scheme under which various *Synopses* are supported by its framework follows the *Java Class Inheritance* pattern as detailed in Section 2.6.3. Thus, any new *Data Summary* type intended to be included to the engine must implement a class that inherits properties from the abstract *Synopsis* class. This way, it is guaranteed that all supported *Synopses* are treated homogeneously by the engine and there is indeed support for fundamental actions (`add, estimate`).

Following this paradigm, we oriented our design towards leveraging *Java Interfaces* to implement the required functionality for capturing a snapshot of an implemented *Synopsis*. This way, *Synopses* could optionally implement it, to enable the *Snapshot* related features on them. The *Java Interface* approach, assures that methods which capture and load a *Snapshot*, are properly implemented by every *Synopsis* utilizing them.

In the process of picking the appropriate format for structuring the *Snapshots* themselves, we assessed numerous techniques that persist *Objects* created during runtime in *Java*. The available techniques were assessed in terms of performance and integration flexibility.

Firstly, and in order to serve the requirement for portability on *Snapshots*, we considered rendering each *Synopsis* object created within the engine, into a *JSON* formatted file. This option though, raised significant concerns regarding the implementation overhead that would be introduced. Developers and engineers would require additional documentation on structuring their inputs and outputs, while the risk for inconsistency would be nurtured under this scheme. Moreover, *JSON* is not natively supported by *Java*, and the usage of it, via external libraries, imposes costly computation steps for parsing files of this format during runtime. Despite *JSON* seemed impractical for serving as the main storage scheme upon which *Snapshots* would be based, we considered it a fit for supporting the *Portability* property we were aiming for. Thus, each *Synopsis* was introduced with a new optional method, `toJSON()`, returning a *JSON* representation of its internal state at a given time moment. This representation is intended to contain both the metadata of a given *Synopsis* (Type, Value & Key Indexes, Stream ID and Dataset Key) and the state itself.

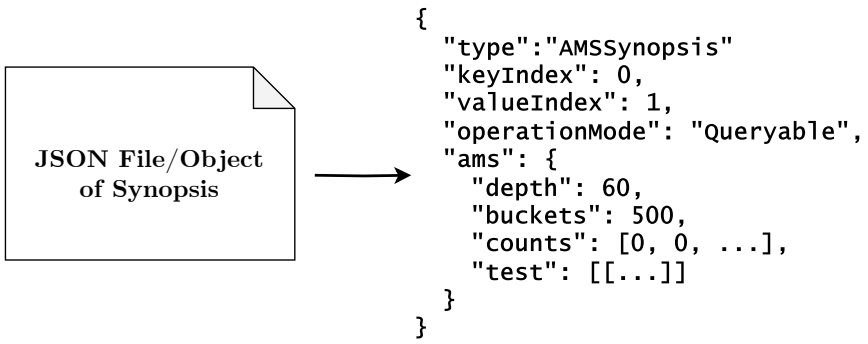


Figure 3.3: Example of *JSON* representation of an *AMSSynopsis*

Let us revisit the issue of the format applied to *Snapshots* handled by the engine. The most prominent option for our case, seemed to be the one that retains information as closely as possible to its original format-namely, a binary format. This prerequisite was established by the need for efficiency and speed in both the capture and retrieval of a *Snapshot*. Our choice ensures that *Synopses* could be reconstructed with minimal computational overhead and without unnecessary complicated transformations or parsing. We moved forward by exploiting the capabilities of *Java's* built-in *Serializable Interface*, which provides a standardized mechanism for serializing objects into files. The scheme utilizes `ObjectOutputStream` and `ObjectInputStream` for handling input and output operations, streamlining the process and ensuring that serialized *Snapshots* can be efficiently written to and restored from external storage. This way, *Snapshots* will be transacted outside the engine context's as files produce upon *byte streams*.

Java Serializable scheme appears particularly advantageous as it offers automatic handling of object graphs, allowing complex structures, including nested objects (arrays, maps or lists) to be serialized without requiring extensive manual intervention, wherever applicable. Since, *SDE* incorporates a wide range of *Synopses*, implemented as, often, complex data structures, the automatic resolution of object graphs introduced by *Serializable* comes in handy, reducing the complexity of aligning current *Synopses* implementations with it.

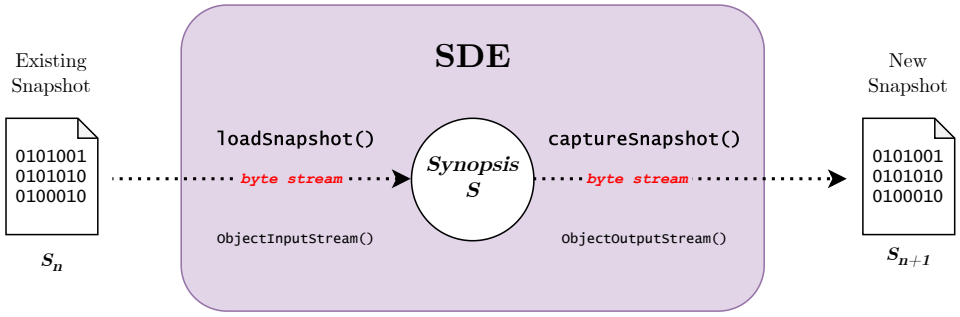


Figure 3.4: Example of serializing/deserializing a *Synopsis* maintained in *SDE* as byte stream.

The adoption of the *Serializable* scheme, aligns our design with *Java's* memory model and garbage collection concepts, making sure that serialized objects retain their integrity and do not introduce unnecessary memory overhead. Our choice of serializing Snapshots via `ObjectOutputStream` and deserializing them through `ObjectInputStream` guarantees that they remain accurate reflections of the in-memory *Synopses* avoiding loss of fidelity.

To conclude, we illustrate in the below class diagram, in Figure, the extensions required to be made to each subclass inheriting the abstract *Synopsis* within *SDE*. These extensions will enable every *Synopsis* currently supported by the engine to become persistent and *Snapshots* of it would be made possible to be captured and reloaded back in.

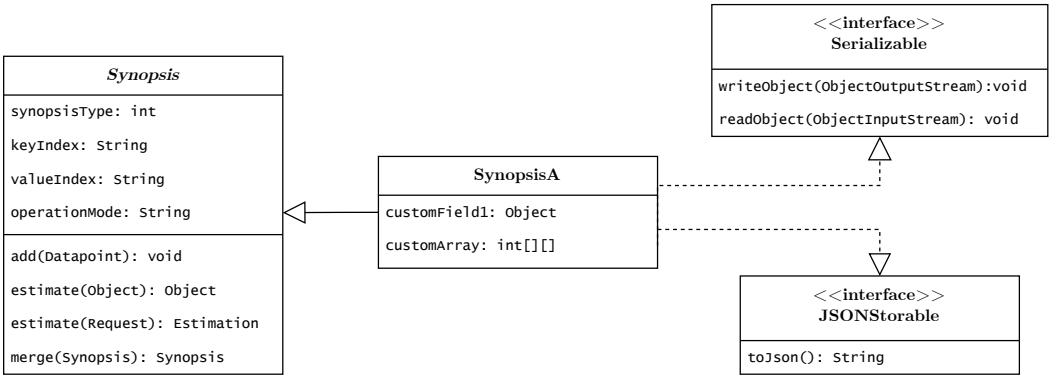


Figure 3.5: Class diagram of a *Synopsis* that is capable of *Snapshot* capturing and retrieval of JSON representation

Earlier before, in Section 3.2.1, we set a prerequisite that parallelization scheme employed in the *Synopses Data Engine* should remain intact after the addition of new features regarding persistency. Towards achieving this aim of ours, we will align the *Snapshot* mechanism such that it guarantees that *Synopses* maintained in parallel workers are also captured and restored in parallel. Subsequently, in this chapter we will address this, by providing the concrete design of the *StorageManager*, crafted in a way encompassing this motive.

3.2.5 Versioning Schemes

Driven by the functional requirements established in Section 3.2.2 and the aforementioned concepts of *Snapshots* we provide a detailed explanation of the versioning schemes which our implementation extends *SDE* to support. Illustrations below showcase those schemes. Notations regarding requests might be present, but we may set them aside for the time being. Request specifications are going to be well presented in the Implementation part, in Section 3.3.

- **Snapshot a Synopsis**

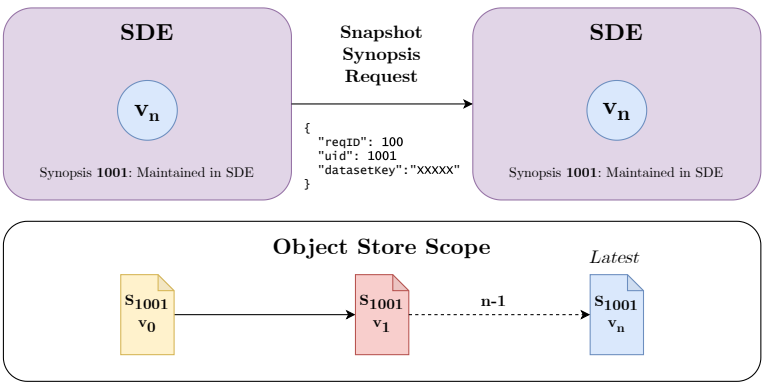


Figure 3.6: *SDE* & *Object Store* Scopes on *Snapshot Synopsis* request

Let v_n be the current state of *Synopsis* 1001 and $n-1$ *Snapshots* of it captured and stored in the *Object-Store*. After a snapshot request arrives, the current state v_n is captured and stored. v_n is marked as the latest snapshot state.

- **Load latest Snapshot of Synopsis**

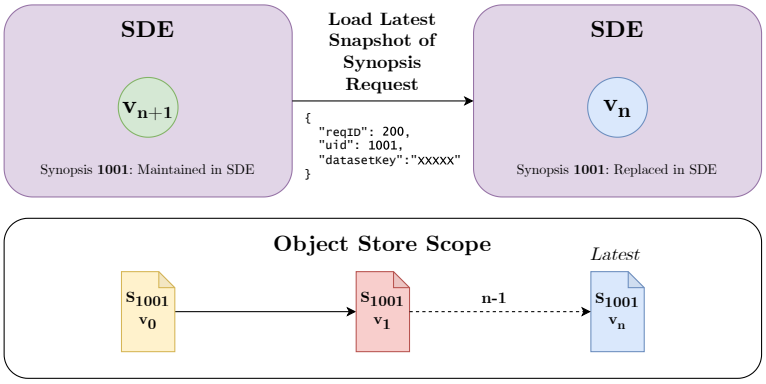


Figure 3.7: *SDE & Object Store* Scopes on *Load Latest Snapshot of Synopsis* request

Let v_{n+1} be the current state of *Synopsis* 1001 and n *Snapshots* of it captured and stored in the *Object-Store*. After a load latest snapshot request arrives, the *Snapshot* v_n overwrites the *Synopsis* maintained in *SDE*.

- **Load specific version of Synopsis Snapshot**

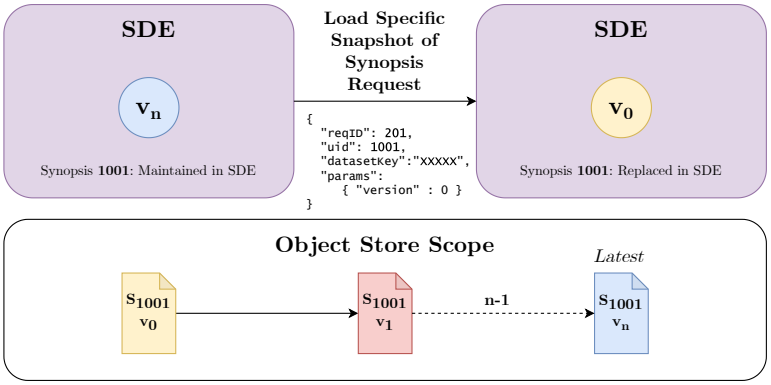


Figure 3.8: *SDE & Object Store* Scopes on *Load Specific Snapshot Version of Synopsis* request

Let v_n be the current state of *Synopsis* 1001 and n *Snapshots* of it captured and stored in the *Object-Store*. After a load specific *Snapshot* version request arrives, requested *Snapshot*, v_0, overwrites the *Synopsis* maintained in *SDE*.

- **Instantiate new Synopsis based on a specific version of Snapshot of other maintained Synopsis**

Let v_n be the current state of *Synopsis* 1001 and n *Snapshots* of it captured and stored in the *Object-Store*. After an instantiate new *Synopsis* upon *Snapshot* request arrives, requested *Snapshot*, v_0, is used to instantiate *Synopsis* 2001 and maintain it in *SDE* alongside *Synopsis* 1001.

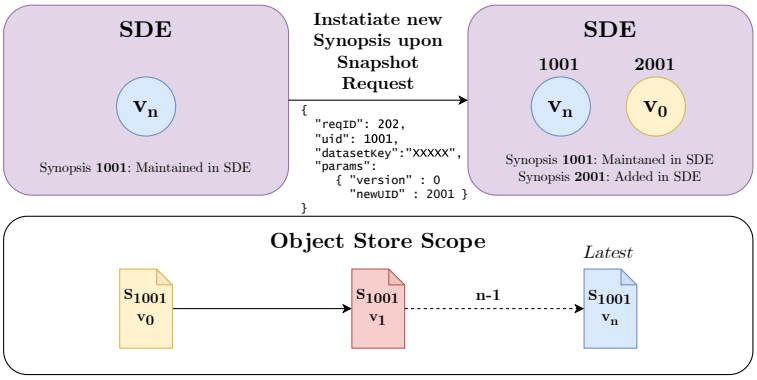


Figure 3.9: *SDE* & *Object Store* Scopes on *Instantiate New Synopsis upon Snapshot* request

3.2.6 Storage Handling

In order to support our design for persistency in *SDE*, detailed in the above sections, a robust and well-structured component is required to implement the set of features related to the handling of *Snapshots* in storage. To serve this purpose, we introduced a new part inside the engine, termed as *StorageManager*.

StorageManager is designed as *static Java Class*, that encompasses methods used to streamline the process of storing and retrieving *Snapshots* from the *Object Store*. Beyond the key operations of saving and loading, the new component is responsible for ensuring integrity and lineage between past and future *Snapshots*.

MinIO and in extend, *Amazon S3*, offer an automatic versioning scheme on stored objects, thus meaning that when an object becomes overwritten within the *Object Store*, the old version is retained and is retrievable. Moreover, lineage tracking is achieved through this embedded versioning, allowing the system to maintain a history of changes made to a stored object. Although this seems as a fit for our case, we decided to move forward by applying a custom versioning mechanism through the aforementioned *StorageManager* component. Our decision was influenced by two factors, which are closely aligned with the non-functional requirements previously established:

1. Compatibility. We are aiming for a design that is not subject to future changes made to the *MinIO* or *Amazon S3* APIs regarding object versions. By implementing custom versioning handling we ensure both forwards and backwards compatibility with the storage providers without the need for future interventions in the code base of *SDE*.

2. Discovery & Integrity. Since *SDE*'s new persistency features aim to facilitate its integration in *STELAR KLMS*, the data layer requirements should be as simple and minimal as possible. The objects required for the *SDE* to operate, should be able to be stored in any bucket, alongside other data without interfeering with them.

Driven by this two objectives, we applied versioning and lineage on the *Snapshots* the *StorageManager* captures and stores, by incorporating a version handling mechanism within it. More specifically *StorageManager* maintains per snapshot *Synopsis* a `.METADATA` file in which it records the *Snapshots* that eventually have been captured in the past. The

`.METADATA` file is JSON formatted, and is created once a *Snapshot*, for a given *Synopsis*, is generated for the first time. The files track information about:

- Information of the *Synopsis* which it is related to.
- The file/object name of each *Snapshot* stored within the bucket and a version number asigned to it.
- The number of the latest version corresponding to *Snapshot* file stored in the bucket.
- The same information maintained for binary *Snapshots* of a *Synopsis*, is also maintained for *JSON* representations that were generated through the `toJSON()` methods.

A sample structure of a `.METADATA` file if illustrated below. The `file_prefix` is calculated upon the `UID` of the *Synopsis* and and the `datasetKey` for which it is being maintained.

```
{
   "file_prefix": "syn_1110_stelar_test_",
   "current_version": "1",
   "type": "AMSsynopsis",
   "versioned_states": {
      "0": {
         "file_name": "syn_1110_stelar_test_v0.json",
         "snapshot_at": "2025-01-24T19:03:33.800592Z"
      },
      "1": {
         "file_name": "syn_1110_stelar_test_v1.json",
         "snapshot_at": "2025-01-24T19:03:37.073573700Z"
      },
   }
   "versions": {
      "0": {
        "file_name": "syn_1110_stelar_test_v0.ser",
        "snapshot_at": "2025-01-24T19:03:33.801198100Z"
      },
      "1": {
        "file_name": "syn_1110_stelar_test_v1.ser",
        "snapshot_at": "2025-01-24T19:03:37.073573700Z"
      },
   }
}
```

The `.METADATA` file is updated every time a new *Snapshot* is captured for the given *Synopsis*. Its creation and updates are handled by the *StorageManager* component which accounts for the integrity of it.

The way our design is fashioned until now, also account for parallelization guarantees. *StorageManager*, as mentioned before, acts as *static Java Class* which can is used within the engine to handle storage related operations. Lets set aside other methods that it may include and shift our focus in the 2 main ones:

- `snapshotSynopsis (Synopsis synopsis, String datasetKey)`
- `loadSynopsisSnapshot (String datasetKey, int uid, Class<T> synopsisClass, int versionNumber)`

These methods are agnostic of the caller and may be used to snapshot parallel instances of a maintained *Synopsis*. In the *Synopsis Data Engine* parallelization is key-based, thus meaning the `datasetKey` of incoming *Datapoints* or *Requests*, for a *Synopsis* maintained in parallel, is appropriately transformed by the *Router* operators as we detailed in Section 2.6.6. By leveraging this feature, *Requests* aiming to perform a *Snapshot* related action for *Synopsis* which was instantiated in parallel, are also processed in parallel. This results to handling a non-unitary number of *Snapshots* in this case. This concept is clearly outlined in Figure, 3.10, illustrated in the next Section (3.2.7).

3.2.7 Design Overview

In the above Sections, we detailed our design on incorporating persistency features within the *Synopsis Data Engine*. Our design focused on making the transition from the initial version of the *SDE* to a version which incorporates stateful *Synopses* and *Snapshot* as smooth as possible. Guaranteeing, firstly, for all the functional and non-functional requirements set in Sections 3.2.1, 3.2.2, the new components' point of attention lies in abstracting the process upon which a *Synopsis* will be able to be captured as a *Snapshot*.

Starting from a flexible way of imposing the required persistency features in each *Synopsis* supported by the *SDE* as of today, the *Java Serializable* interface is leveraged to support the conversion of `Synopsis` objects, created and maintained during the runtime, to binary files through byte streams. This binary representation of a `Synopsis` object forms, partially, *Snapshot* which can later be used to rollback the *Synopsis* itself to the captured state or, even instantiate new *Synopses* in the engine upon that. A *Snapshot* also combines the capture of *JSON* representation of the *Synopsis* at a given moment, enabling this way a portability potential.

The complex task of handling *Snapshots* storage-wise is assigned to a new component incorporated in the *SDE*, the *StorageManager*. *StorageManager* exploits the efficient and flexible features of the *MinIO Object Store* in order to establish a storage layer for the *Snapshots*. Towards maximizing compatibility (forward and backward), our new component transacts with the storage layer by using *Amazon S3 SDK* to ensure seamless integration to other storage providers supporting the S3 scheme. With the vision of implementing the *StorageManager* as a robust and long-live component of the *SDE* we followed a custom versioning scheme for *Snapshots*, handled internally, avoiding grounding our implementation on the available object versioning plans built-in into *MinIO*, which are subject to special configurations and potential incompatible updates.

The proposed design is agnostic of parallelization or of the environment in which it runs on. This fact enables its seamless integration into the key-based parallelization scheme employed in *SDE*. As a part of the *Dataflow Operators* that are used to maintain *Synopses*, the *StorageManager* component may be used to capture *Snapshots* in parallel for *Synopses* that are maintained in such a way.

A summary of our design is illustrated below. For simplicity's shake we represent the parallel computations in abstract parallel *SDE* box.

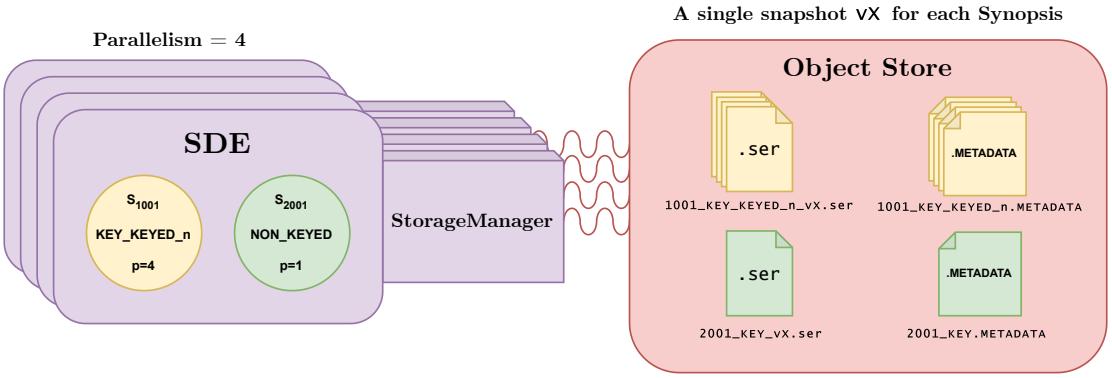


Figure 3.10: An abstract view of the persistency design where a single snapshot, denoted v_X, for each maintained *Synopsis* has been captured.

3.3 Implementation

Towards surrounding our design inside the *Synopsis Data Engine*, interventions on the *Flink Dataflow* (2.6.6) were required. In this Section we trace thoroughly modifications and additions made to *SDE*'s codebase after it was refactored to fit our needs. We will organize the following sections such that each of them goes through changes made per *Dataflow Operator* and additions made mainly concerning the *StorageManager* class.

3.3.1 The StorageManager Class

The *StorageManager* as mentioned in the previous Section, is the main component handling the entire set of operations related to storage, ranging from storing a serialiazed version of a *Synopsis* to in the *Object Store* or loading another one back into the engine. Besides the main features for I/O, the *StorageManager* also creates and updates the structure of the `.METADATA` file(s), which is maintained per *Synopsis*, used to track the stored *Snapshots* inside the storage layer and trace lineage between them.

Since the storage provider of our choice, *MinIO*, emulates the *Amazon S3 API* we moved forwarding by using the official *S3 SDK* released and maintained by *Amazon Web Services*. This way, it ensured that the *SDE* will be able to migrate to another S3 storage provider with minimal modifications. The *StorageManager* class, is used by every parallel worker running the *SDECoFlatMap* in the local *JVM* [1]. The class instantiates a static `awssdk.services.s3.S3Client` object once, during the initialization. This object, is the main component for interacting with the *Object Store* and handles the connection. By instantiating it only once, we prevent unnecessary overhead during request processing later, caused by authorization and connection requests if no `S3Client` object was already in place. In the below class diagram a simplified view of the *StorageManager* class is presented. For the full class diagram please refer to the Appendix A of this Thesis

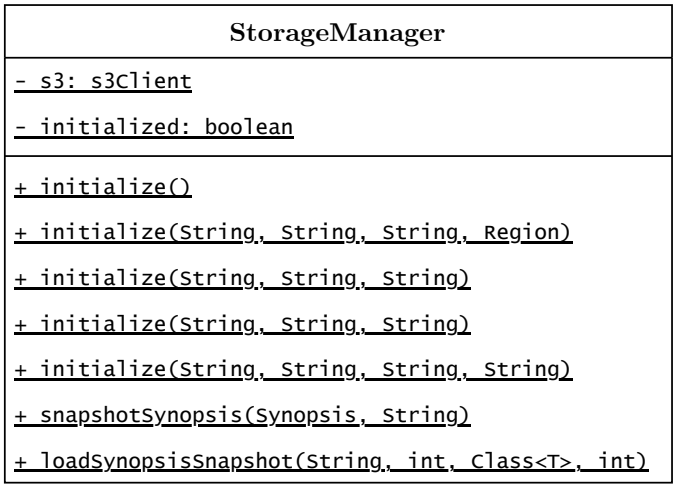


Figure 3.11: Minimal Class Diagram of the *StorageManager*

It is noticeable that the class provides a set of initialize methods. Each of them, supports on the one hand a different storage provider (*Amazon* or *MinIO*) and on the

[1] Each *TaskManager* in *Apache Flink* runs a local Java Virtual Machine which accomodates Tasks.

other hand a different authentication scheme per provider. Later on, in the next Chapter, we will address the way the *MinIO* instance, serving as the data layer in the *STELAR KLMS*, authenticates requests in the lake's multi-user environment and how the data layout is protected through *OpenID* [2] and how our *StorageManager* is already aligned with the applied authentication scheme.

The `snapshotSynopsis()` and `loadSynopsisSnapshot()` methods are used by the *SDECoFlatMap* operator when the respective operations are requested through the extended *Streaming API*. Since these methods are called within the *Maintenance Operator* it is possible that they may be used in parallel, capturing this way `#parallelism` number of *Snapshots* for a *Synopses* which is maintained in parallel.

3.3.2 The SDECoFlatMap Operator

Back in Section 2.6.6, we outlined the purpose of the *SDECoFlatMap* operator within the *SDE's* dataflow model. The operator acts as the core of the engine, maintaining *Synopses* on-demand. In setups where *SDE* is executed on a Flink Cluster deployed with non-unitary parallelism, *SDECoFlatMap* runs in every parallel worker, supporting this way the parallelization scheme that SDE is designed with. Our main point of intervention lies within this operator.

The operator hosts two `flatMap()` methods, each dedicated to processing either `Datapoints` or `Requests`, depending on the case. The `flatMap2(Request)` method is responsible for implementing the *Streaming API* and, based on the *Request* type and *Synopsis* type, it invokes the corresponding method to execute the appropriate logic. Driven by this architecture, we extended the *API* to support new request types that emerge from the functional requirements and schemes, outlined in Sections 3.2.2 and 3.2.5. The following requests were added in `flatMap2()` of the operator:

- **Request** `100`: Snapshot a *Synopsis*.

```
{
   "requestType": 100,
   "datasetKey": "AAPL",
   "uid": 1001
}
```

- **Request** `200`: Load the latest *Snapshot* of a running *Synopsis* onto it.

```
{
   "requestType": 200,
   "datasetKey": "AAPL",
   "uid": 1001
}
```

- **Request** `201`: Load a selected *Snapshot* version of a running *Synopsis* onto it.

```
{
   "requestType": 201,
   "datasetKey": "AAPL",
   "uid": 1001,
   "params" {
       "version": "4"
   }
}
```

[2] OpenID is an open authentication protocol that allows users to log in to multiple services using a single identity, eliminating the need for multiple credentials.

- **Request** 202: Instantiate a new *Synopsis* based on *Snapshot* of another one.

```
{
    "requestType": 202,
    "datasetKey": "AAPL",
    "uid": 1001,
    "params" {
        "version": "4",
        "newUID": "2001
    }
}
```

The handling clause for each of this requests has access to the local `HashMaps` that are maintained on every parallel *SDECoFlatMap* instance, containing the objects of *Synopses* mapped to keys. This way it has control and access over maintained *Synopsis* during runtime. Depedending on the *Request* and the *Synopsis* that it references (if found), the appropriate clause within the `flatMap2()` invokes a static method from the *StorageManager* class which eventually returns a *Synopsis* object (*Load Requests*) or a boolean to acknowledge for success or failure.

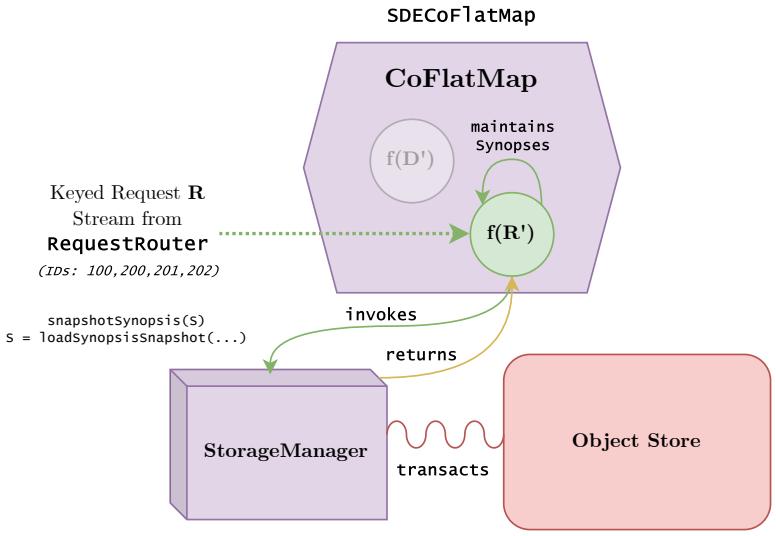


Figure 3.12: Flow of actions when handling storage-related *Requests* in the *SDECoFlatMap*

To further streamline the integration of *SDE* into *STELAR KLMS*, we included a fifth request type in the *Streaming API* that changes the credentials with which the `S3Client` is initialized, without the need to bring down the *Flink Job*.

Request `101`: Change the credentials of the `S3Client`

```
{
    "requestType": 101,
    "datasetKey": "xxxxxxx",
    "uid": xxxxxxx,
    "params":{
        "type":"sts",
        "access_key": "AXDBMCADSFJFASD",
        "secret_key": "eJIJWeLAMXLXMJDMAKSJDapldkAD",
        "session_token": "eyKkfKADFAKSD.eOIAsdkk..."
    }
}
```

3.3.3 The DataRouterCoFlatMap Operator

Since our new features include, partially, the instantiation of new *Synopses*, our implementation should also apply the parallelization scheme of the *SDE* on tuples destined to those *Synopses*, such that they operate seamlessly. The operator in charge of manipulating the *Datapoints* destined to *Synopses*, maintained in parallel is the *DataRouterCoFlatMap* as detailed in Section 2.6.6.

In that case, a request intended to instatiate a new *Synopsis* derived from a *Snapshot* of an already existing one, should be treated similarly with a classic maintenance request. Thus, the modification applied in the operator was straight forward. A small difference occurs regarding the handling of the requests: the `UID` upon which the partitioning settings will be generated is the one defined in the parameters when dealing with the `202` *Request*.

3.3.4 SDE Persistency Implementation Overview

Conclusively, now that we succintly defined the steps followed to enhance the *Synopses Data Engine*, we have equipped it with strong and versatile persistency features that will also pave the way for the following Chapter.

We introduced the *StorageManager* class which abstracts storage related operations and provides the *Flink Dataflow* with flexible features for handling I/O transactions with the underlying *Object Store*. The enhancements added in the *SDECoFlatMap* operator introduced new request types, allowing dynamic snapshotting, loading, and instantiating *Synopses* upon *Snapshots*. Furthermore the engineer gains efficient control over the data-layer when provided with the ability to alter the *StorageManager* settings on the fly. Additionally, the modifications applied to *DataRouterCoFlatMap* operator ensure that new *Synopses*, instantiated from *Snapshots*, integrate seamlessly into the existing parallelization scheme.

Below we illustrate the persistency implementation on *SDE*.

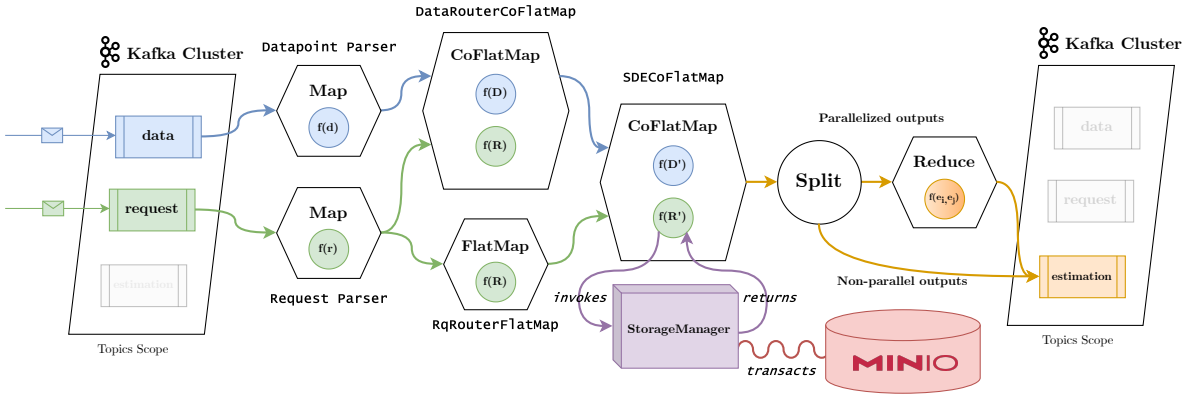


Figure 3.13: Persistency Implementation in the *Synopsis Data Engine*

Chapter 4

Integration

This final chapter brings this Thesis to a close as it assembles our design for persistency in the *Synopses Data Engine* providing a holistic perspective of the vision upon which we began our journey. Our aim on integrating *SDE* as a stateful data summarization tool in the *Knowledge Lake* of *STELAR* will be fulfilled by introducing *Interoperability* as another fundamental property of it.

4.1 Preliminaries

4.1.1 Tool Execution in STELAR KLMS

Back in Section 2.7.2, we highlighted the capability of *STELAR KLMS* on executing *Workflows* incorporating several steps of processing applied to the *Lake's* data, by leveraging *Data Analytics & Interlinking Tools* or even *LLMs*. A rich *Workflow Execution API* bundles the interaction between the data analyst and the execution engine of the *STELAR* core.

Tools may be executed within the *STELAR's* context as *Tasks* of broader *Workflow*. Since data analytics or LLMs can get computationally intensive or require specialized hardware (CUDA-Enabled GPUs etc.), the *API* provides support for two execution modes, *in-cluster* and *remotely*. *STELAR* is usually deployed in a *Kubernetes Cluster* [26]. By leveraging the rich *API* provided, it is capable of executing tools as *Kubernetes Jobs* ensuring seamless execution cycle management. Tools are required to be packed into *Docker Images* in order for *STELAR* to accommodate their in-cluster execution.

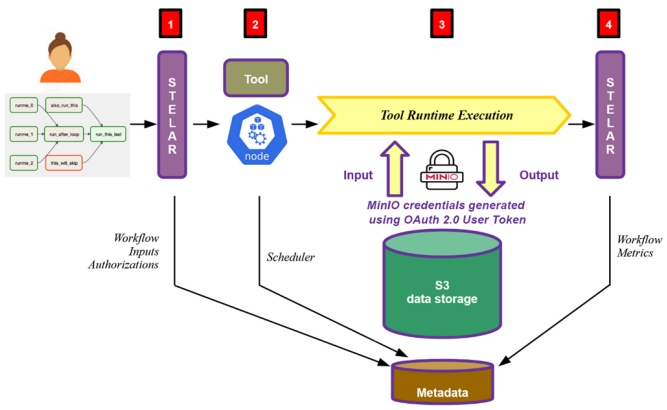


Figure 4.1: Execution flow of a tool in *STELAR KLMS* (in-cluster mode)

4.1.2 Roadmap of Integrating SDE into STELAR KLMS

When it comes to *SDE*, packing it into a single *Docker Image* seemed impractical or even impossible. Additionally, *SDE*, as a streaming-based application, requires external control at all times and will not operate standalone. Our focus shifted in the request oriented nature of the engine, and an idea emerged for implementing a *Client SDK* which would be used to stream requests into the engine and control it remotely. Thus, *SDE* would be integrated into *STELAR* in a hybrid mode, where the flexible and lightweight *Client* is executed as volatile *Kubernetes Job* in the cluster, and the engine itself is running in a robust and stable *Flink Cluster*, possibly deployed within the same *Kubernetes Infrastrucure*. The *Client SDK* not only makes the integration of *SDE* in the Lake much more feasible and practical, it also may be utilized in scenarios where the engine is running standalone or is incorporated as a part of other systems.

4.2 The need for Metadata

Synopses Data Engine was initially designed as a component, offering a *Streaming API* to interact with. The lifetime of the maintained *Synopses* was limited to the lifecycle of the *Flink Job*. The features added to it, facilitating persistency, have encompassed its ability to maintain non-volatile *Synopses* in a storage layer. Despite that, *SDE* provides limited metadata about its internal state during runtime. *Flink's Queryable State* was initially used to provide introductory information about the maintained *Synopses* during runtime. As this feature is subject to deprecation in *Flink's* latest versions, a work-around is required in order to gain permanent and robust access to the engine's state. Moreover, in the context of *STELAR* the *Queryable State* feature would be impractical to support through due to various factors.

On the other hand, the *Streaming API* alone does not provide guarantees for the proper handling of a received request, limiting this way the error handling when the *SDE* runs in enclosed environments and access to the output of the *Flink Job* is not possible.

Combining the two aforementioned bold points, that were, by that moment, limiting the engine's capabilities, we decided to incorporate a second output other than the *Estimations*. This new *Logging Stream* is able to emit *Messages* that either represent *Responses* for ingested *Requests* or *Responses* that contain information about the currently maintained *Synopses*. This way we serve a hybrid purpose: the need for metadata, that later are registered in the *Data Catalog of STELAR KLMS* and the need for error handling.

4.2.1 Design & Implementation

The entire volume of information regarding the *Logging Stream* is at any given moment maintained or produced by the *SDECoFlatMap* operator. Our main aim was not to interfere with the original output stream in order to maintain guarantees for the produced *Estimations*. Since we could gain access to our information of interest from a single operator we decided to attach a *Side Output Stream* to the maintenance operator. This choice of ours, disclosed a great shortcoming of the *CoFlatMap* operator, which is not able to host side output streams and provides no access to *Context*. In order to tackle this problem, the *CoFlatMap* operator was replaced with a *CoProcessFunction* one. Their main difference lies in *CoProcessFunction's* ability to provide access to keyed or non-

keyed streams with independent, stateful processing logic for each input stream along with timers and a *Context*. The new operator imposes no implementation modifications other that replacing the old *CoFlatMap* one with it. Integrating it, in the *Dataflow* of *SDE* was straightforward with no complications occurring on the way.

A new side output stream was attached to the new maintenance operator and now *Messages* have been made possible to be emitted without interfering with the *Estimation Stream*. It was expected that *Messages* could be produce in parallel, thus meaning that for one request transmitted with non-unitary parallelism, more than one response *Messages* will be produced. An appropriate *Reducer* operator has been fashioned to eliminate this problem and produce a single *Response*, no matter the parallelism degree of the *Request*. Non-parallel *Responses* go through the splitter and are directed to the topic without being reduced. The new design applied to the *Dataflow* of *SDE* is illustrated in Figure 4.2

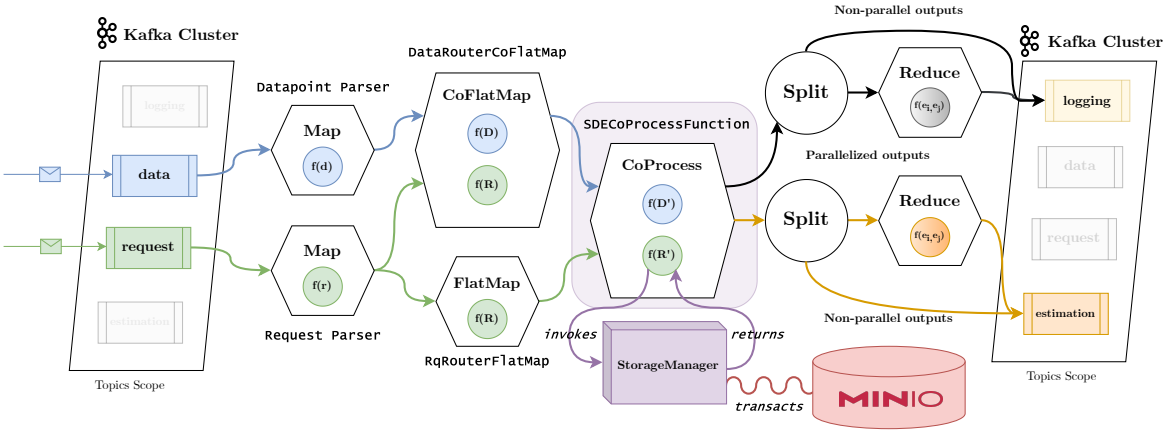


Figure 4.2: *SDE Dataflow* with persistency features and logging output stream

Responses emitted in the *logging stream* may be intercepted by any consumer within a specified timeout period, effectively simulating a request-response API interaction.

4.3 Synopses Data Engine Python SDK Client

The challenging task of performing requests and feeding data into the *Synopses Data Engine* under a uniform way, was addressed by introducing a *Python Client SDK* as a wrapper to the *Streaming API* of the engine.

The scope under which the *Client* was designed, intended to treat the extended *SDE's API*, offering the logging stream, in a way that emulates a classic request-response API. Moreover, it contains features that facilitate the engine's integration into *STELAR KLMS*. The library of available operations ranges from actions relevant to *Synopses'* maintenance, update or querying to newly featured persistency operations used for capturing snapshots, loading them into in the engine or altering the credentials used to access the storage layer.

4.3.1 Features

In order to streamline the process of developing processing flows incorporating the *Synopses Data Engine* the *Client* replicates a local model of the entities lying within the engine. The developer has access to a local `Synopsis` class that acts as a bundler for all methods that can be applied to a maintained *Synopsis* within the *SDE*. For explanatory purposes we are referring to every object instatiated from the *Synopsis* class as *Proxy Object*. Specifically the following methods are available through the `Synopsis` class:

- `Synopsis.new(args)`: Create a local Proxy object that is used to simulate a *Synopsis* locally. It may initiate any action that can be performed on a running Synopsis in the SDE.
- `Synopsis.add(args)`: Create a local Proxy object of a *Synopsis* and add in the *SDE*. The object may be used for further interaction.
- `Synopsis.estimate(args)`: After a Proxy object is successfully in sync with the SDE (through `new()` or `add()`), an estimation may be received.
- `Synopsis.snapshot(args)`: After a Proxy object is successfully in sync with the SDE (through `new()` or `add()`), a snapshot of the running *Synopsis* mays be captured.
- `Synopsis.loadLatestSnapshot(args)`: After a Proxy object is successfully in sync with the SDE (through `new()` or `add()`), the latest snapshot of the running *Synopsis* may be loaded back in the engine to overwrite it.
- `Synopsis.loadSnapshotOfVersion(args)`: After a Proxy object is successfully in sync with the SDE (through `new()` or `add()`), a specified snapshot of the running *Synopsis* may be loaded back in the engine to overwrite it.
- `Synopsis.newSynopsisFromSnapshot(args)`: After a Proxy object is successfully in sync with the SDE (through `new()` or `add()`), a specfied snapshot of a *Synopsis* may be used to instantiate a new *Synopsis* in the engine, identical to the initial one.

Besides the `Synopsis` class, a `Client` class acting as connector to the *Kafka Topics* is available and abstracts the entire process of implementing a connection with the cluster with appropriate producers and consumers while also providing an interface with method templates ready destined to reach the *SDE's Streaming API*. In terms of capturing responses, the *Client* maintains a queue in which it places consumed log messages that may be subject to answer previously sent *Requests*. In terms of universal handling methods the client offers the following:

- `Client.__init__(args) -> Client`
- `Client.consumeMessage(args) -> dict`
- `Client.sendStorageAuthRequest(args)`
- `Client.sendRequest(args) -> dict`
- `Client.sendDataPoint(args)`
- `Client.close()`

4.4 SDE into STELAR KLMS

4.4.1 Deployment

Since *SDE* may run on an any *Flink Cluster*, when packed as a *JAR* file, we honored the flexibility and efficacy of the *Flink Kubernetes Operator* [27] to set up *Flink* inside the core deployment of *STELAR KLMS*. This approach not only alleviates data analysts and developers from the burden of complex local deployment configurations but also ensures the automated and seamless management of the *Flink Cluster's* lifecycle, thereby facilitating the smooth operation of *SDE* on it. Moreover, *Flink* is complemented by a *Kafka Cluster* running by its side, maintaining the ingestion and digestion topics required for *SDE*. By keeping *SDE* as close as possible to the *data layer* of the engine not only eliminates configuration complications but also minimizes response times and prevents underperformance events which may be occurring due to network bottlenecks during data transactions.

Conclusively, by leveraging the *Client SDK's* capabilities, *SDE* was deployed as is in *STELAR's Kubernetes Cluster*, without the need for complex configurations applied by the end-users. The *Client SDK* can support the creation of *SDE Workflows* ranging from a simple set of analytic steps to a composite chain of computations that are run through the engine. Developers may pack their scripts, incorporating *SDE* for processing (via the client), into lightweight *Docker Images* that later on can be used to create *Task Executions* within the context of *STELAR Workflows*.

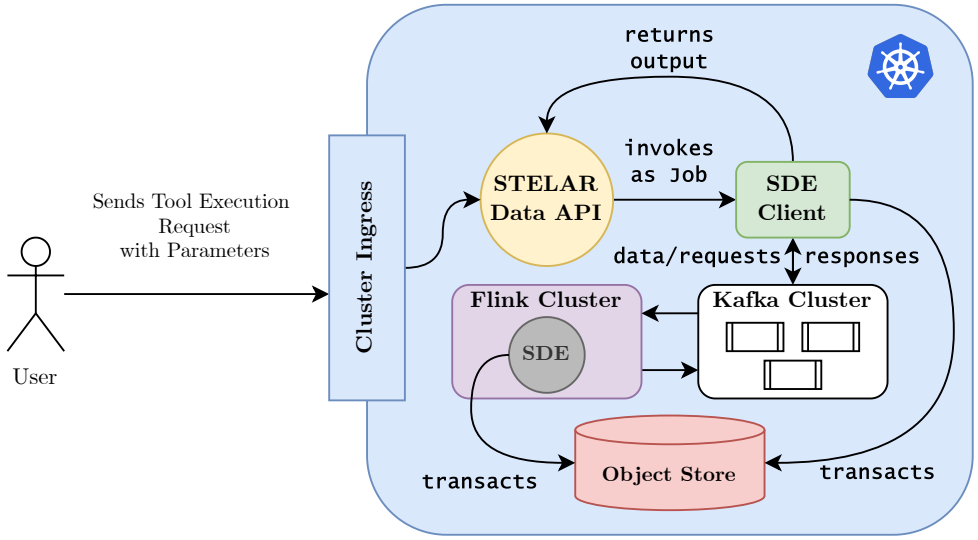


Figure 4.3: *SDE* deployment within *STELAR KLMS*, abstract view

.

Using this approach the *Synopses Data Engine* became a stateful stream summarization component of the *STELAR KLMS*, enhancing the *Lake's* versatile toolkit with a composable and lightweight summarization framework.

Chapter 5

Conclusions

This Thesis addressed the topic of enhancing the *Synopses Data Engine* with new features, for incorporating persistency on capturable *Snapshots* of *Synopses* and enhancing its interoperability. By incorporating the snapshot mechanism, the engine now supports the retention and retrieval of *Synopses* over time, enabling historical tracking of insights on summaries, that were previously ephemeral and volatile. Furthermore, improvements applied to the engine's *Streaming API*, enhanced the observability capabilities, encouraging this way, its seamless connection to external data ecosystems.

The newly added features facilitated the integration of the engine to a broader data ecosystem, the *STELAR Knowledge Lake*, introducing this way additional merit to the lake's capabilities. This design not only validated the feasibility of the enhancements but also highlighted the importance for the presence of stream summarization components in knowledge-driven environments handling vast amounts of raw data.

By designing and implementing a persistent, observable, and seamlessly integratable summarization component, this work contributes not only to the advancement of *SDE* itself, but also to the broader sector of real-time data analytics enhanced with stateful properties.

Building on the foundation, this work has laid, even more sophisticated features may be added. Concepts for automatic capturing of snapshots can be fashioned and designed upon the establishment of algorithmic thresholds, that mark the decision point at which the state of a given *Synopsis* is considered outdated in comparison with the latest captured snapshot. Additionaly, snapshots themselves may play an essential role in restoring the engine's state on fault-recovery scenarios or be leveraged to guide new federation schemes, upon which geo-dispersed instances of the *Synopses Data Engine* are interlinked and synchronized through snapshot exchange.

Appendices

Appendix A

Software Specifications

A.1 SDE Code Base

The source code for the new version of *Synopses Data Engine* supporting persistency and interoperability may be found in the below repositories.

Synopses Data Engine

- SDE Personal Repository
  `https://github.com/petroud/SDE-E`
- SDE Repository in STELAR-EU organization
  `https://github.com/stelar-eu/Synopses-Data-Engine`
- SDE Repository in SoftNET TUC organization
  `https://github.com/softnet-lab/SDE`

On the other hand, the source code for the *SDE Python SDK* may be found in the below repositories, while the client remains installable via `pip` using the package name `sde-py-lib`.

SDE Python Client SDK

- SDE PySDK Personal Repository
  `https://github.com/petroud/SDE-py-lib`
- SDE PySDK in STELAR-EU organization
  `https://github.com/stelar-eu/SDE-py-lib`
- SDE PySDK Repository in SoftNET TUC organization
  `https://github.com/softnet-lab/SDE-py-lib`

A.2 SDE Streaming API

In the below table, the requests supported by the *Streaming API* of SDE are outlined. More detailed documentation may be found in the repositories denoted in part A.1 of this Appendix

ID	Operation	Description
1	**ADD-K**	Maintain a new synopsis in the engine with KEYED partitioning
2	**DELETE**	Delete a currently maintained synopsis
3	**ESTIMATE**	Estimate a queryable synopsis
4	**ADD-R**	Maintain a new synopsis in the engine with RANDOM partitioning
5	**ADD-C**	Maintain a new CONTINUOUS synopsis
100	**SNAPSHOT**	Capture a snapshot of a maintained synopsis into the object store
101	**AUTHENTICATE WITH S3**	Send a request with S3 STS credentials and endpoint to configure the StorageManager
200	**LOAD LATEST SNAPSHOT**	Retrieves the latest snapshot of a synopsis (If previously generated) onto the running instance of it.
201	**LOAD CUSTOM SNAPSHOT**	Retrieves a specific version of snapshot of a synopsis (If previously generated) onto the running instance of it.
202	**NEW SYNOPSIS FROM SNAPSHOT**	Retrieves a specific version of snapshot of a synopsis (If previously generated) and instantiates a new synopsis inside the engine of the same type. Returns the UID of the new synopsis
1000	**GET ALL SYNOPSES BY KEY**	Returns all maintained synopses from the engine grouped by dataset key

A.3 StorageManager Class Diagram

A full class diagram of the *StorageManager* component, featured in the snapshot mechanism may be found below.

Bibliography

[1] Wieger R. Punter, Odysseas Papapetrou, and Minos Garofalakis. *OmniSketch: Efficient Multi-Dimensional High-Velocity Stream Analytics with Arbitrary Predicates*. 2023. arXiv: 2309.06051 [cs.DB]. URL: https://arxiv.org/abs/2309.06051.

[2] Rudi Poepsel-Lemaitre et al. "In the land of data streams where synopses are missing, one framework to bring them all". In: *Proc. VLDB Endow.* 14.10 (June 2021), pp. 1818–1831. ISSN: 2150-8097. DOI: 10.14778/3467861.3467871. URL: https://doi.org/10.14778/3467861.3467871.

[3] Antonis Kontaxakis, Nikos Giatrakos, and Antonios Deligiannakis. "A Synopses Data Engine for Interactive Extreme-Scale Analytics". In: *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. CIKM '20. New York, NY, USA: Association for Computing Machinery, Oct. 19, 2020, pp. 2085–2088. ISBN: 9781450368599. DOI: 10.1145/3340531.3412154. URL: https://doi.org/10.1145/3340531.3412154 (visited on 06/06/2024).

[4] *Apache Flink Framework*. Version 1.9. 2020. URL: https://flink.apache.org/.

[5] *STELAR Project*. Funded by the European Commision, HORIZON Europe, GA No. 101070122. 2022-2025. URL: https://stelar-project.eu/.

[6] Kontaxakis Antonios. "Design and implementation of a distributed synopsis data engine on Apache Flink". Available at https://doi.org/10.26233/heallink.tuc.85602. MSc Thesis. Chania, Crete: Technical University of Crete, 2020.

[7] *Apache Flink DataStream Operators*. Version 1.19.0. 2024. URL: https://nightlies.apache.org/flink/flink-docs-release-1.19/docs/dev/datastream/operators/overview/.

[8] *Apache Kafka Framework*. Version 2.4. 2024. URL: https://kafka.apache.org/.

[9] Philippe Dobbelaere and Kyumars Sheykh Esmaili. "Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper". In: June 2017, pp. 227–238. DOI: 10.1145/3093742.3093908.

[10] M.Garofalakis et al. *Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches*. Vol. Vol. 4, Nos. 1–3 (2011) 1–294. 2012. URL: http://dx.doi.org/10.1561/1900000004.

[11] Graham Cormode and S. Muthukrishnan. "An improved data stream summary: the count-min sketch and its applications". In: *Journal of Algorithms* 55.1 (2005), pp. 58–75. ISSN: 0196-6774. DOI: https://doi.org/10.1016/j.jalgor.2003.12.001.

[12] Moses Charikar, Kevin Chen, and Martin Farach-Colton. "Finding frequent items in data streams". In: *Theoretical Computer Science* 312.1 (2004). Automata, Languages and Programming, pp. 3–15. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(03)00400-6.

[13] Rasmus Pagh, Morten Stöckel, and David P. Woodruff. "Is min-wise hashing optimal for summarizing set intersection?" In: *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '14. Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 109–120. ISBN: 9781450323758. DOI: 10.1145/2594538.2594554. URL: https://doi.org/10.1145/2594538.2594554.

[14] Burton H. Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692. URL: https://doi.org/10.1145/362686.362692.

[15] *Apache DataSketches*. Version 6.2.0. 2024. URL: https://datasketches.apache.org/.

[16] *StreamLib: A Library for Streaming Algorithms*. Version 2.9.5. 2016. URL: https://github.com/addthis/stream-lib.

[17] *streaminer: Java library for stream mining algorithms*. Version 1.1.1. 2020. URL: https://github.com/mayconbordin/streaminer.

[18] Do Le Quoc et al. "StreamApprox: approximate computing for stream analytics". In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. Middleware '17. Las Vegas, Nevada: Association for Computing Machinery, 2017, pp. 185–197. ISBN: 9781450347204. DOI: 10.1145/3135974.3135989. URL: https://doi.org/10.1145/3135974.3135989.

[19] Shivnath Babu and Jennifer Widom. "Continuous queries over data streams". In: *SIGMOD Rec.* 30.3 (Sept. 2001), pp. 109–120. ISSN: 0163-5808. DOI: 10.1145/603867.603884. URL: https://doi.org/10.1145/603867.603884.

[20] Antonios Kontaxakis et al. "And synopses for all: A synopses data engine for extreme scale analytics-as-a-service". In: *Information Systems* 116 (2023), p. 102221. ISSN: 0306-4379. DOI: https://doi.org/10.1016/j.is.2023.102221. URL: https://www.sciencedirect.com/science/article/pii/S0306437923000571.

[21] Raghu Ramakrishnan et al. "Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 51–63. ISBN: 9781450341974. DOI: 10.1145/3035918.3056100. URL: https://doi.org/10.1145/3035918.3056100.

[22] *MinIO Object Storage*. Version RELEASE.2025-01-20T14-49-07Z. 2025. URL: https://min.io/.

[23] *Keycloak IDP*. Version 25.0. 2024. URL: https://www.keycloak.org/.

[24] *Amazon S3*. 2025. URL: https://aws.amazon.com/s3/.

[25] *Celebrate Amazon S3's 17th birthday at AWS Pi Day 2023*. 2023. URL: https://aws.amazon.com/blogs/aws/celebrate-amazon-s3s-17th-birthday-at-aws-pi-day-2023/.

[26] *Kubernetes*. Version 1.29. 2024. URL: https://v1-29.docs.kubernetes.io/docs/home/.

[27] *Flink Kubernetes Operator*. Version 1.10. 2024. URL: https://nightlies.apache.org/flink/flink-kubernetes-operator-docs-stable/.