


Tailored IoT & BigData Sandboxes and Testbeds for Smart,
Autonomous and Personalized Services in the European
Finance and Insurance Services Ecosystem



D5.3 – Library of Parallelized Incremental Analytics - III

| | |
|---|--|
| Revision Number | 3.0 |
| Task Reference | T5.2 |
| Lead Beneficiary | LXS |
| Responsible | Ricardo Jiménez-Peris |
| Partners | LXS, GLA, CTAG |
| Deliverable Type | Report (R) |
| Dissemination Level | Public (PU) |
| Due Date | 2021-12-31 |
| Delivered Date | 2022-03-30 |
| Internal Reviewers | NUIG, HPE |
| Quality Assurance | INNOV |
| Acceptance | WP Leader Accepted and Coordinator Accepted |
| EC Project Officer | Beatrice Plazzotta |
| Programme | HORIZON 2020 - ICT-11-2018 |
|  | This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement no 856632 |

Contributing Partners

| Partner Acronym | Role ¹ | Author(s) ² |
|-----------------|-------------------|---|
| LXS | Lead Beneficiary | Ricardo Jiménez-Peris |
| LXS | Contributor | Boyan Kolev Javier Pereira Luis Miguel Garcia Jesús Manuel Gallego Pavlos Kranas Jose Maria Zaragoza |
| GLA | Contributor | Richard McCreddie, Craig Macdonald, Iadh Ounis |
| CTAG | Contributor | Andrea Becerra |
| NUIG | Internal Reviewer | Martin Serrano |
| HPE | Internal Reviewer | Alessandro Mamelli |
| INNOV | Quality Assurance | John Soldatos |

Revision History

| Version | Date | Partner(s) | Description |
|---------|------------|------------|---|
| 0.1 | 2022-03-01 | LXS | ToC Version |
| 0.2 | 2022-03-02 | All | Add section 4 |
| 0.3 | 2022-03-02 | LXS | Extends conclusions |
| 1.0 | 2022-03-02 | LXS | Finalize the document for internal review |
| 1.1 | 2022-03-16 | HPE | Internal review |
| 1.2 | 2022-03-29 | NUIG | Internal review |
| 2.0 | 2022-03-26 | LXS | Submitted for internal QA |
| 2.1 | 2022-03-29 | INNOV | Internal QA |
| 2.2 | 2022-03-30 | LXS | Finalize the document |
| 3.0 | 2022-03-30 | LXS | Version ready for the submission |

¹ Lead Beneficiary, Contributor, Internal Reviewer, Quality Assurance

² Can be left void

Executive Summary

The goal of Task T5.2 “Incremental and Parallel Data Analytics” is on one hand to deliver a set of algorithms that can be used in the finance and insurance sector and can be considered incremental and be parallelized in order to improve the overall performance and on the other hand, to provide the enablers for those algorithms to be executed incrementally. In addition, those algorithms should be used by a data analyst to extract information as close to real-time as possible. Typical algorithms for these types of scenarios can be found in the area of frequent pattern mining, time series prediction analysis, collaborative filtering, and others. Even if they have been widely adopted from applications in the aforementioned sectors, they usually rely on static data that has been persistently stored in a datastore, and as a result, even if they can be parallelized, they cannot be considered as incremental.

As the INFINITECH platform provides an innovative data management layer that claims to overcome the inherited and existed barriers for correlating data *at-rest* with streaming data, it provides a unified data framework for integrated query processing on both types of sources. The latter makes use of a streaming engine that provides additional operators that allows this correlation of data and relies on the basic pillars of the data management layer of the platform. In the scope of T5.2, firstly we rely on the work that has been carried out in the corresponding tasks of WP3 that implements that layer, and on the results of T5.3 that provides *online aggregations*. By exploiting the advancements of those tasks, we are in a position to re-design popular algorithms used in the finance and insurance sectors, implement them in a distributed manner so that they can be easily scaled out and serve very high rates.

Another important objective of task T5.2 is to provide incremental analytical processing that can be exploited by such algorithms of the finance and insurance section. Towards this, the internal storage engine of the data management layer, the INFINISTORE, has been re-designed in order to provide the enablers of such type of analytics. This took place during the second phase of task T5.2, as in the first phase, the storage engine had to be extended to provide support for the *online aggregates*, the basic pillar of the technology that is being provided under the scope of T5.3. With the advancements of T5.2 in this second and final phase, the storage engine can now propagate data modifications to its upper layers, which are being consumed by the INFINISTORE’s API. At first, it could provide *incremental scans*, which is the basic operation for incremental analytics. Now it is also possible to enhance all relational algebraic operations (such as filters, projections and aggregations) supported by the storage engine with incremental analytics.

This deliverable describes how we take advantage of INFINITECH’s existing tools and frameworks in order to parallelize time series algorithms for correlation discovery and forecasting, a popular family of algorithms that are being used in a variety of use cases in the finance sector regarding risk assessment for stock or retail trading. Our library can be executed in parallel and the results are being returned incrementally. Moreover, it describes how the incremental analytics can be used in practice by such algorithms or can be integrated with other technology components of INFINITECH, such as the Streaming Processing Framework and the Semantic Interoperability Engine. This report summarizes the work that has been carried out during all phases of the project (M05-M27). It has been further extended with the complete documentation of the incremental analytics engine and the additional work that has been carried out in the last phase of the project, with the implementation of the Kafka source connector that makes use of the incremental analytics that have been already integrated into the main product of LeanXcale, its datastore, or the INFINISTORE, as widely known in the project.

Table of Contents

- 1 Introduction..... 6
 - 1.1. Objective of the Deliverable..... 7
 - 1.2. Insights from other Tasks and Deliverables..... 8
 - 1.3. Updates from the previous version (D5.2) 9
 - 1.4. Structure..... 9
- 2 Parallel and Incremental algorithm for time series analytics..... 10
 - 2.1 Methods 10
 - 2.1.1 Time series correlation discovery..... 10
 - 2.1.2 Time series forecasting..... 11
 - 2.2 Potential use cases in various sectors 12
- 3 Enablers for Incremental Analytics..... 13
 - 3.1 Motivation 13
 - 3.2 Traditional Analytical Processing..... 14
 - 3.3 Incremental Analytical Processing..... 15
 - 3.4 Documentation..... 17
 - 3.5 Incremental scans in Practice 20
- 4 Kafka source connector 25
 - 4.1 Core implementation of the source connector 25
 - 4.2 Kafka source using filters..... 27
 - 4.3 Kafka source using aggregations 28
 - 4.4 Kafka source connector in practice 29
- 5 Conclusions..... 31
- 6 References..... 34

List of Figures

- Figure 1: Example of a pair of time series that the method found to be highly correlated over the first several sliding windows of 500 time points, but not thereafter. 10
- Figure 2: Example of a time series (red) and its top correlates (green) discovered by the method..... 11
- Figure 3: A typical query plan 14
- Figure 4: Class diagram of direct-API for incremental scans 17
- Figure 5: Class diagram of the Kafka source connector 26
- Figure 6: Class diagram of filter conditions in kafka..... 28
- Figure 7: Class diagram of aggregations in kafka 29

Abbreviations/Acronyms

| | |
|------|--|
| AI | Artificial Intelligence |
| API | Application Programming Interface |
| DDoS | Distributed Denial of Service |
| DL | Deep Learning |
| FFT | Fast Fourier Transformation |
| HTAP | Hybrid Transactional and Analytical Processing |
| iSAX | Indexable Symbolic Aggregate Approximation |
| ML | Machine Learning |
| PAA | Piecewise Aggregate Approximation |
| RNN | Recurrent Neural Network |
| SVD | Singular Value Decomposition |
| WP | Work Package |

1 Introduction

Modern enterprises tend to use data coming from a variety of heterogeneous sources that are collected via numerous means and usually stored in a data warehouse or a data lake. Data analysts make use of sophisticated Artificial Intelligence (AI) algorithms for Machine Learning/Deep Learning (ML/DL) in order to extract valuable information that is crucial for the business intelligence of the organization. Focusing on the finance and insurance institutions, typical use cases include online risk assessment through the correlation discovery of stocks or other financial products, online fraud detection of a financial transaction, optimal resource management of the overall portfolio of a customer, which can be either a person or another business institution, and potential identification of investment opportunities.

Typical uses of such cases usually rely on historical data that has been imported into a data warehouse from an operational datastore or a stream of events or other IoT data. The reason for migrating data from one data source to another stems from the inherent barrier of performing analytics over an operational datastore, due to the competitiveness of these two different types of workloads, as explained in the corresponding deliverables of task T3.1 (“Framework for Seamless Data Management and HTAP”). Moreover, performing analytics over a stream of data introduces another obstacle, as sophisticated AI algorithms usually require the full scan of a huge amount of data that cannot be maintained in memory. As a result, modern architectures require the migration of data coming from various sources to a data lake, which will allow the data analysts to execute their algorithms in the complete dataset in a parallelized manner.

Migrating data from one source to another (i.e. from an operational datastore to a data warehouse) requires complex architectures in case there is the need for analytical processing of (near) real-time data. Those architectures, apart from being complex, are very difficult to maintain and come with their drawbacks as they do not holistically solve the problem. To overcome those issues, in INFINITETCH we envision the Intelligent Data Pipelines that has been described in the corresponding deliverables of T3.4 (“Automated Parallelization of Data Streams and Intelligent Data Pipelining”), which makes use of the INFINISTORE as the target datastore that data is being continuously migrated to it. By exploiting the innovations of INFINISTORE that has been developed within the project, typical AI algorithms used in the finance and insurance sector can be redesigned and make use of such enablers to overcome the current technology barriers in data management systems.

However, those algorithms tend to rely only on static data that has been stored in a persistent storage medium. They usually require a pre-processing that takes place in the data management layer, typically by submitting a query statement that is being pushed down to the datastore and retrieve results that will be further used as an input for their processing. Moreover, the results they get are related to the snapshot of the dataset at the time query was received by the datastore. As modern architectures usually migrate data snapshots periodically (in terms of batches or micro-batches) to another datastore that will receive and process the incoming analytical workload, this is not considered a problem, as there is this inherent barrier we cannot cope with. In INFINITETCH however, its data management layer provides Hybrid Transactional and Analytical Processing (HTAP) capabilities and due to the additional innovations such as its *online aggregates* implemented under the scope of T5.3, we remove the need for migrating data to a data warehouse. This means that AI algorithms can submit analytical queries on the live dataset that is being continuously updated with data ingested from various sources.

As AI algorithms can now retrieve results over a dataset that is being modified in the run-time, we can do better than that. In this task T5.2, we implement the *incremental analytics* that can be used as the enablers for incrementally designing those algorithms. This means that a deployed analytical processing framework can submit a query once, get the results for its initial analysis, and continuously receive a data feed of modifications that can be fed to its algorithms and update the results on the run time, without having to periodically re-submit the same query to get the data results that reflect the current snapshot of the dataset, at the new point in time when the query was resubmitted. This can be of a huge benefit for a variety of AI algorithms in the finance and insurance sector that their purpose is to trigger notifications for

alerts or potential opportunities. This can be used in risk assessment for bank investment, anti-money laundering or fraudulent financial transactions.

Being able to use *incremental analytics* and submit a query statement once in the datastore and then retrieve the results continuously, without having to periodically re-submit the query and retrieve the same results slightly modified, have a significant impact both from a business and from a technical perspective. Regarding the technical perspective, the execution can be much more efficient due to the minimization of the data that need to be transmitted over the network: the results are being retrieved once and only data modifications will be now transmitted. Moreover, the infrastructure will consume lesser CPU cycles and will require lesser memory both in data management and in the analytical processing layer as the main computational power will be spent only at the time the query is submitted. From a business perspective, the data analyst or business developer can have alerts or notifications at the time a change occurred, rather than having to wait for the periodic execution of the algorithm.

Another important objective of the work that is being carried out under the scope of task T5.2 is to parallelize the identified popular AI algorithms in order to boost their performance. Parallelizing however an algorithm does not guarantee that the performance will be improved by default. There are the capabilities for advanced analytical processing that enables such a performance boost, which can be exploited by an alternative design for the execution of such algorithms instead. These enablers for advanced analytics consist of a family of technology components such as the *online aggregates* (developed under the scope of T5.3) and the incremental analytics that are being reported here. Additionally, under the scope of T3.4, the INFINITECH platform provides the framework for automated parallelization of data streams. This enables the aforementioned AI algorithms to be deconstructed and deployed in different streaming processing nodes. These nodes can be parallelized and in fact, they can scale out automatically with no downtime by exchanging their internal state during run-time. Moreover, this framework can also benefit from the *incremental analytics* that are being presented in this report and consume data feeds coming from data modification operations propagated by the persistent database level. The feeds can be later fed into the parallelized steaming operators of the streaming processing framework, which can be used by the AI algorithms themselves. As a result, the re-design of these popular target algorithms takes into account the holistic approach for the data management layer of INFINITECH.

To conclude, Task T5.2 “Incremental and Parallel Data Analytics” aims to leverage a set of typical algorithms and libraries used for artificial intelligence in the insurance and finance sectors that can benefit from the unique characteristics of the data management layer of INFINITECH, in order to re-design them and deliver parallelized taking into account the incremental analytics, and all other innovations and offerings of the integrated data management layer of the project. A first family of such analytics had been studied and reported in the first version of this document. In this final version, we additionally full document the implementation of the incremental analytics of the INFINISTORE, along with the implementation of the Kafka source connector that makes use of them. It is important to highlight that the incremental analytics have been merged into LeanXcale main product while the Kafka source connector is an additional product that allows LeanXcale’s client to use a Kafka broker and retrieve data from the datastore in an incremental and non-bounded manner.

1.1. Objective of the Deliverable

The objective of this deliverable is to report the work that has been done in the context of task T5.2, at the final phase of the project (M27). This last version of this deliverable that has been released, extending and modifying when necessary the content of this document. As new technological advancements coming from the other technical tasks have been now delivered, the work on this task was planned to be further developed at this last phase, as it relied on the basic pillars of the data management layer. The work that has been delivered during the first and second phases of the project (M05-M20), was mainly focused on one hand on the implementation of the *incremental scans* that are the basics for the delivery of *incremental analytics* and on the other hand, on the design of the parallelization of a time series algorithm

for risk assessment prediction, which can be also used in other domains. For this purpose, we studied how to deliver this algorithm in a parallel fashion, by experimenting with the results of other tasks of the project. Finally, this deliverable reports on how we can deploy such types of algorithms to make use of streaming processing in order to deliver results incrementally. In the last phase, we enhance the incremental capabilities in order to provide continuous queries using such analytics in all algebraic operations supported by the INFINISTORE internal storage engine. Finally, we additionally implemented the Kafka source connector to INFINISTORE that makes use of the incremental analytics.

1.2. Insights from other Tasks and Deliverables

The work that has been carried out in the scope of T5.2 relies on the outcomes of the T2.1 ("User Stories and Analysis of Stakeholders' Requirements") that define the overall user stories and requirements of the use cases of INFINITECH, and how the implementation can be integrated with the achievements of the other technical tasks, as defined in the INFINITECH RA. All those are part of WP2, and more precisely of T2.3 ("Specification of Enhancements to BigData & IoT Platform") and T2.5 ("Open Banking APIs, Testbeds and Data Assets Specifications"). Apart from this, WP3 gives significant input to this task, as it implements the basic technological pillars for T5.2 to rely on. WP3 provides the overall data management layer of the project. More precisely, T3.1 ("Framework for Seamless Data Management and HTAP") provides the ability to rely on a single operational datastore, the INFINISTORE, that can be used to process analytical workload without having to migrate data to a data warehousing technology. T3.3 ("Integrated Querying of Streaming Data and Data at Rest") implements the unified data query processing framework, which allows the correlation of streaming with batch processing. Moreover, T5.3 ("Declarative Real-Time Data Analytics") provides the implementation of online aggregations that can be used by the algorithms in this task. The online aggregations allow the pre-calculation of an aggregated value, thus removing the necessity to perform a full scan on a data table to calculate that value when this is needed. These technologies can be used as enablers to allow the efficient parallelization of commonly used AI algorithms in the financial and insurance sectors. Moreover, this parallelization can take advantage of the parallelization of the streaming processing nodes provided by T3.4 ("Automated Parallelization of Data Streams and Intelligent Data Pipelining"). Additionally, task T3.4 can also consume data feeds coming from *incremental analytics* that are being currently developed and reported in this document. The data feed will reflect the data modifications on the dataset stored in the operational database and can be used as an input to the parallelized streaming processing nodes, that are further used by the AI algorithms. Last but not least, the data stream of the transaction logs produced by the data modification operations can also feed the *streaming plugins* developed within the context of the Semantic Interoperability Framework developed under the scope of T4.1 ("Shared Semantics for BigData and IoT Streams") and T4.2 ("Massive Distributed Processing of Semantically Linked Streams"). This framework internally makes use of a triple store that need to be populated with data coming from a persistent storage that needs to be inter-exchanged with other datasets. As a result, the *incremental scans* can update this internal triple store, via the Intelligent Data Pipelines of task T3.4 so that the semantic engine can be always up to date.

We can see that this task is tightly coupled and interconnected with the majority of other tasks that belong to the technical work packages of WP3, WP4 and WP5 and provide the technological building blocks and offerings of the INFINITECH ecosystem. Therefore, we prioritized at the first phase of the project to give more focus on these specific tasks first, which will be exploited at this second phase by T5.2. Moreover, the implementation of the *incremental analytics* requires the provision of *incremental scans* first, which rely on the propagation of the data modifications by the storage engine of INFINITECH. However, a parallel work needed by T5.3 to implement the *online aggregates* also required the extension and modification of the storage engine, and more precisely, its indexing mechanism and data structures. Therefore, it was not possible to progress on both tasks at the same time and we prioritized at the first phase to give more focus on the T5.3. Right now, the implementation of the *incremental analytics* has been finalized.

1.3. Updates from the previous version (D5.2)

This is the last version of this deliverable. It includes the addition of section 5 which provides the documentation of our Kafka source connector that makes use of the innovation that was developed during the first and second phase of this task, along with a demonstrator of its deployment and use.

1.4. Structure

This document is structured as follows: Section 1 introduces the document. Section 2 provides details on how a commonly used AI algorithm can be parallelized and benefit from using the innovations provided by the data management layer of INFINITECH. Section 3 focuses on the incremental analytics part: it firstly states the motivation behind such technology and then it provides the basic principles and design of their implementation. It adds documentation for its use and provides a concrete example with a demonstrator relying on a real use case taken from pilot#2 of INFINITECH. Section 4 provides the documentation of our Kafka source connector implemented under the scope of this task and additionally includes a demonstrator of its. Finally, section 5 concludes the document.

2 Parallel and Incremental algorithm for time series analytics

Nowadays, we are witnessing the production of large volumes of complex data, often in the form of time series. These may originate from various sources, e.g. financial activities, such as stock trading or bank transactions, as well as monitoring network activity or collecting data from sensors. Time series analytics allow extracting useful insights from streams of numeric data through various statistical, algorithmic, or machine learning methods.

2.1 Methods

Intending to leverage the inherent features of the INFINITECH data management layer, which is currently implemented under the corresponding tasks of WP3, focusing on aspects such as its increased scalability, the allowance for data ingestions at very high rates, and the online aggregations, under the scope of this task, we are currently focusing on two aspects of time series analytics: correlation discovery and forecasting methods discussed briefly below.

2.1.1 Time series correlation discovery

Time series correlation discovery aims at finding similarities across time series, as shown in Figure 1, based on a distance metric, most commonly the Euclidean distance. This can be achieved using various methods for dimensionality reduction, e.g. singular value decomposition (SVD) [1], Fast Fourier Transform (FFT) [2], [3], wavelets [4], piecewise aggregate approximation (PAA) [5], random projections [6], as well as indexing, e.g. the iSAX (indexable Symbolic Aggregate Approximation) tree index [7], locality sensitive hashing through sketches [8] and more.

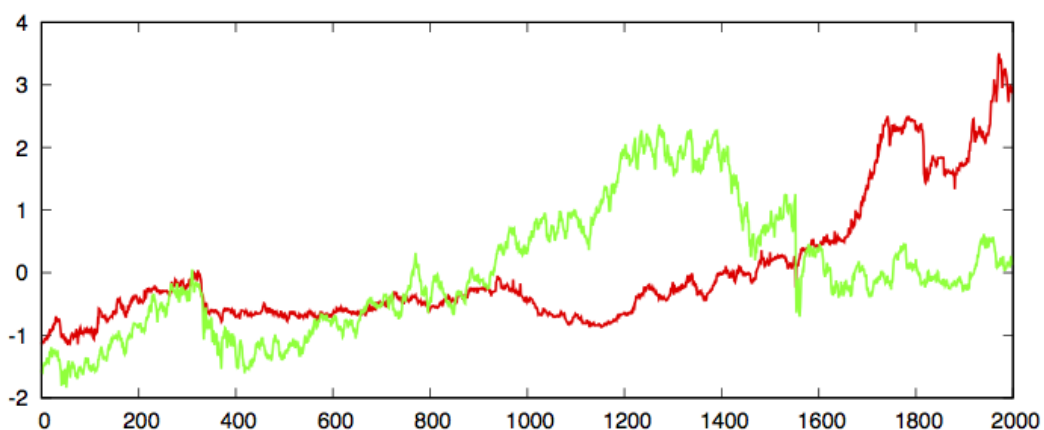


Figure 1: Example of a pair of time series that the method found to be highly correlated over the first several sliding windows of 500 time points, but not thereafter.

For the INFINITECH project, we concentrate on ParCorr, a recently introduced parallel incremental approach for fast correlation discovery over windows of time series data [8]. The method scales to millions of parallel time series and achieves 95% recall and 100% precision. To address dimensionality reduction that nearly preserves the Euclidean distances across the latest window of time, it uses a random projection approach to compute in an incremental manner and in parallel the sketch of each time series. The sketch is a tiny structure that summarizes the time series by preserving its locality, so simply comparing sketches provides filtering to significantly reduce the search space for the much more expensive comparison of time series. We are focusing on that approach, taking into account the overall data management layer of the INFINITECH platform, which will exploit the use of the central data repository in order to boost the performance of this locality sensitive hashing approach. This will allow for almost real-time discovery of the top-k correlates of a given time series (see Figure 2), which further helps to build a model for instantly predicting future values of the time series as a function of the values of its correlates.

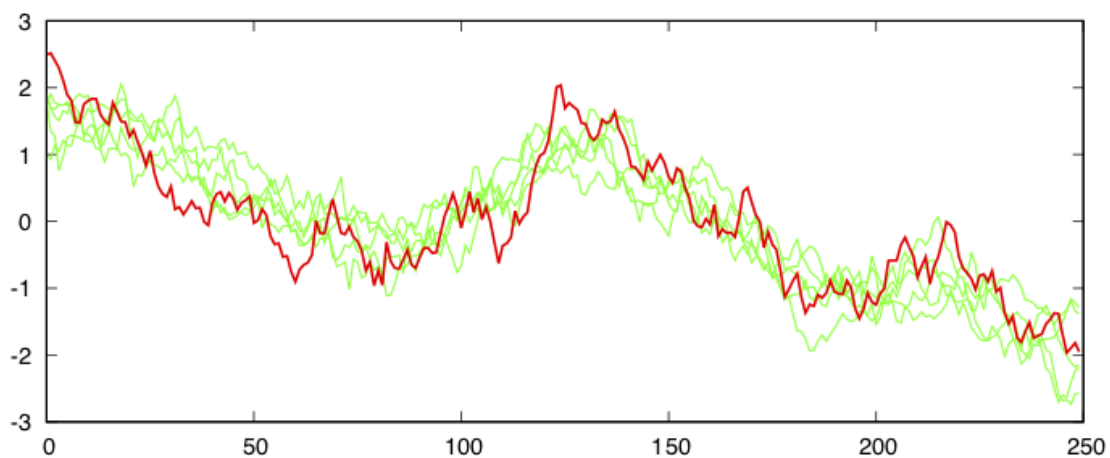


Figure 2: Example of a time series (red) and its top correlates (green) discovered by the method.

2.1.2 Time series forecasting

Statistical methods for time series forecasting [9] usually analyse the values of a single time series, often with a focus on the most recent ones, to predict the next value. Simple methods, such as moving averages (MA), i.e., the average of the last few slices of time, in many cases provide a good approximation for the expected value of the time series in the future moments. In other models of approximations that need to give more weight to the most recent values, exponential smoothing [10] would improve the forecasting accuracy. In more sophisticated scenarios, ARMA methods or deep recurrent neural networks (RNNs) can be used to capture more complex dependencies in data. Very often, all these methods work better when applied to the differences between any two consecutive points in a time series instead of directly to the time series itself, a transformation known as “differencing”.

For the INFINITECH project, we propose an efficient framework, benefiting from the features of its integral data management layer, to incrementally perform in real-time *moving averages*, *exponential smoothing*, and *differencing* of each time series towards predicting its values for the upcoming time points. Moreover, different methods can be combined to capture specific behaviour, e.g. similarity search can be done by computing sketches on top of exponentially smoothed values and/or differencing and/or moving averages.

2.2 Potential use cases in various sectors

Time series analytics has applications in many different domains, some of which are mentioned below:

- Finance/stock trading: a pair of time series (say Google and Apple prices) that were similar before, but have ever since diverged, may represent a trading opportunity.
- Seismology: correlated signals from several different but not much distant seismic sensors may suggest that they are all related to the same seismic event.
- Network monitoring: similar traffic patterns from different sources may indicate an attempt for distributed denial of service (DDoS) attack.
- Retail/trading: future demand of a product can be approximated with statistical methods for forecasting (using the recent history of sales and/or seasonal factors), as well as through the use of discovered correlations with other indicators (such as prices and sales of other products, weather conditions, social trends, etc.).

3 Enablers for Incremental Analytics

This chapter is related to the second major objective, which is the implementation of the enablers for incremental analytics. We will give an example taken from one of the pilots of the project to highlight the motivation of such a technology, and then we will compare how the query is typically processed traditionally by the INFINISTORE, and how it is processed with the implementation of the incremental analytics. Then, the documentation and a demonstrator of its usage will be given.

3.1 Motivation

The basic principle behind incremental analytics is the ability to retrieve results as soon as they become available. That means, the moment when data arrives the data management system. With a traditional approach, the data user or application developer submits a query statement (that can be a SQL or other) to the datastore targeting a dataset, the database examines which part of the target dataset satisfies the query and returns this part as the result. In most common cases the result is a virtual table or a list of arrays, but this is not a necessity. This result set is being used then by a part of a micro-service or a building block of an application or is being given as the input for an AI algorithm. The important thing that needs to be highlighted here is that the result set is the part of the dataset that satisfies the initial submitted statement, at the point in time when the query was actually submitted to the datastore and the latter started its processing. This means that if the data user or application developer wants to re-evaluate the data and check what might be the result after a specific time period, he or she would need to re-submit the same query and get the updated result, which will now reflect the snapshot of the dataset at this new point in time when it was re-submitted.

A typical example coming from the finance sector is the evaluation of the risk assessment for investing in various products. We will take the example of pilot#2 (“Real-time risk assessment in Investment Banking”) of INFINITECH that deals with such scenarios. Pilot#2 receives from various data streams the financial currencies of different products per second. This data stream is being ingested to the data management system of the project, the INFINISTORE, which persistently stores it to a data table. Then, an AI algorithm is being periodically executed to calculate the value at risk of the products and propose possible opportunities to potential investors. The algorithm needs to consume a big amount of data to be able to calculate with a specific accuracy the value at risk. Typically, it requires collecting the financial currencies of the latest months, when each month contains $60 \times 60 \times 24 \times 30 = \sim 2.5$ million records per product. This would require for the database to firstly process its entire dataset and validate that ~ 2.5 million records that satisfy the query, then transmit all this information through the network, and finally, the AI risk assessment algorithm to further process all these records and return with a decision. As this requires a significant amount of processing power and time, the algorithm is being executed periodically every 5 minutes.

From our example, it is obvious that the proposed identified opportunity to the potential investor is taking into account the snapshot of the database the moment the AI algorithm requested data from the datastore. The next identified opportunity will rely on an updated snapshot, 5 minutes later the first one. However, pilot#2 is continuously receiving financial currencies for products, every second. With our traditional approach, we will always miss the $5 \times 60 = 300$ intermediate snapshots between the two points in time when we periodically request data. What if we could have all intermediate information? This would mean that we need to evaluate the submitted query in every snapshot of the dataset. Thus, this would mean evaluating the query whenever the database transits its state due to the execution of a data modification operation. This is exactly what the incremental analytics are solving.

3.2 Traditional Analytical Processing

But how does this work? Let's think about a traditional relational database that implements a standard JDBC interface to allow data connectivity. In JDBC, the application developer would have opened a new *connection*, then would have created a *statement* based on an SQL query, then would have executed that *statement* and finally would have been returned by the database a *result set*. The latter is the structure or instance of the class that contains and handles the data that satisfy the query and will be returned to the invoker. As we are discussing how the JDBC works, in Java the *ResultSet* is eventually extending the *Iterator* interface, and as such, it is practically an iterator. Each iterator, no matter what its internal implementation would be, declares a *next* method. We should now imagine the returned data as a list of arrays, where the array can be the data row of the virtual table defined by the SQL statement. In each invocation of the *next* method, firstly the execution is being done *synchronously*, and as such, this method is a *blocking* method. Secondly, the result of the invocation is the next item in our list of arrays, or a *null*, if we have reached the end of the list.

To better understand how execution and processing of a query are being done, we need to also deep down to the query engine level. The query receives the query statement that needs to be executed via the JDBC connection that has been opened from the client/driver. It then compiles it in order to transform the statement that has been received in a text form, to a structure form that can be manipulated or processed by the machine. This is the query plan. Most modern query engines send the initially generated query plan to a query *optimizer*, whose role is to produce different equivalent query plans, by applying specific transformation rules. Then it calculates the overall cost of execution of each plan, and finally, using a *greedy* type of algorithm, it returns the optimal one in terms of cost, which will be finally executed. The query plan is a tree of operations that all together formulate the overall plan for the execution of the submitted query. Taken from D3.2 ("Hybrid Transactional/Analytics Processing for Finance and Insurance Applications – II") an example query plan can be the one depicted in Figure 3.

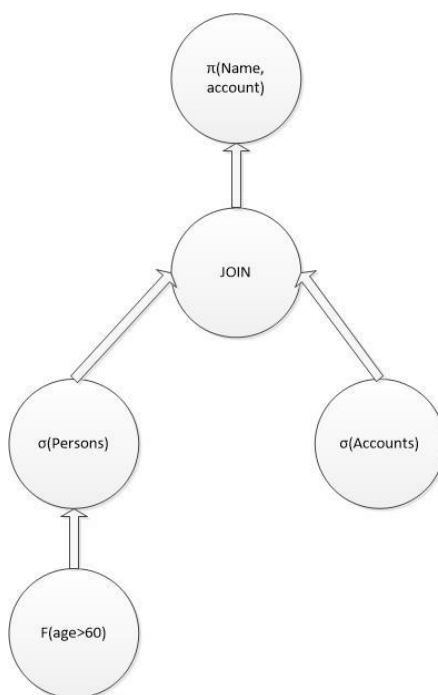


Figure 3: A typical query plan

The query contains the JOIN between two tables (Persons, Accounts), while the data user applies a filter criteria over the table *Persons*, while finally, we *project* two columns (Name, Account) that define the resulted virtual table to be retrieved. The query plan contains different types of query operators, and in our example, we can see the *filter*, *scan*, *join*, *projection*. Internally the query engine contains one or more

implementations per each type of *operation*, while each *operation* extends the interface *iterator*, thus, implements the method *next*. The query engine instantiates an instance of each of the *operations* that take part in the overall query plan. The query plan is now formulating a data pipeline of query operators that can be now executed. This is also known as the *volcano iterator model*.

Now, how does the driver collect the results that can be now accessible from the application code or AI algorithm? The reader needs to recall that in JDBC, the result of the execution of a statement is a *ResultSet*, that is an *Iterator* and such, implements the method *next*. Moreover, as we saw all query operations involved in the query plan to be finally executed are *iterators*, and such, implement the method *next*. As a result, each time the application code or AI algorithm invokes the method *next* of the *ResultSet*, the latter will invoke the corresponding one of the upper-level query operators of the tree of the query plan. In our example, that will be the *projection*. This one will invoke the corresponding *next* method of the query operation of the lower level, and so on. At the lowest level of the query tree, there lies the type of operations that access the *storage* or the *index* and retrieve data bytes that are related to a data row in a data table. The data row then is being forwarded through the data pipeline to the upper layers (as the lava being flown from down to earth up to the volcano), where it is being processed accordingly, and finally, it reaches the data user through the initial invocation of the *next* method of the *ResultSet* of the driver. This is how the query processing is being done in the query engine of the INFINISTORE, and generally, this is how most modern database management systems have been designed.

At this point, we need to underline that this was a simplified example, and the query processing is more complicated. All query operations do not request for the *next* value *synchronously*, but instead, they keep internally an array of rows that can be forwarded. This is considered as a *buffer*. Operators fill this *buffer* immediately when there is a row available so that the upper layer does not need to wait. A similar approach happens between the communication between the driver and the backend, the query engine: results are filling an intermediate buffer, and are being sent via batches to the driver. In each invocation of the *next* method, the driver gets the next item of the buffer, and when the buffer is closed to empty, it requests the next batch from the query engine, that has been filled by the upper operation of the query tree. However, using a buffering mechanism or not, the basic principles are the same.

3.3 Incremental Analytical Processing

As we saw in the previous subsection, the invocation of the *next* method returns the next item of the list of the data rows. Or *null*. This happens when there is no other item to be retrieved from the lower layer, or the storage or index. When the method returns a *null* method, the operator should be closed and eventually removed by the garbage collector. This happens because the query execution is taken over a fully *timely bounded* dataset. If we think of a dataset as the collection of incremental transitions between states that have been created by the invocation of a data modification operation, then the dataset is a live object that is continuously being changed. However, with the traditional analytical query processing, we can only see a snapshot of the living dataset: the one at the point in time when the query was submitted for execution. That is what we call a fully *timely bounded* dataset.

In incremental analytics, the query statements are being executed over a timely unbounded dataset, which is a dataset that is not a snapshot but rather can change as time progress. These are now continuous queries, whose results are being returned continuously or as we better call *in an incremental manner*. What does this mean in practice? Let's imagine that we have the following statement that returns all records from the *Persons* table, whose name starts with 'P':

```
SELECT *
FROM PERSONS
WHERE name LIKE 'P?'
```

Let's assume that this data table has 1000 rows, and 100 of them have a value in the *name* column whose first letter is P, like Pedro, Paula or Pelopidas. In the traditional approach, we would have received those 100 rows and then the *iterator* would have been closed. However, using incremental analytics, we submit a continuous query over a *timely unbound* dataset and the corresponding *iterator* never closes (or at least, it closes after a pre-defined period of time). During the execution, the *iterator* returns the same 100 rows as before, however, that time, it stays open. This means that the query is continuing to evaluate the dataset, as the latter transits its state, reflecting the results of the data modification operators. In practice, whenever a new row is being inserted, it will be evaluated against the continuous query.

In our example, after the initial return of the first 100 rows, the *iterator* is blocked and waits for new items to be sent from the backend, the query engine. In the meantime, new data arrives and we add a new record in the *Persons* table, whose value of the column *name* is Samantha. Samantha starts with 'S', it will be evaluated by the query statement, and won't be picked up. Then a new record arrives, with the value as 'George'. Again, it will be evaluated and won't be selected. Later on, a new record is being ingested to the database, whose value is now 'Patricia'. This will be picked up by the lower query operator and further pushed to the upper layers of the query tree. In our example, there are no other layers, so this data row will be sent to the driver, and eventually, the *iterator* of the *ResultSet* will return the value and wait for the next one to arrive.

Let's examine the same with our previous example, whose query tree was depicted in Figure 3. The query was the following:

```
SELECT P.name, A.account
FROM PERSONS P INNER JOIN Account A ON P.ID=A.P_ID
WHERE P.age>60
```

Here we want to see all accounts of persons that are above 60 years old. The execution returns an initial list of 100 data rows. However, even if the query is more complicated, the *iterator* remains open, and so does the query execution plan with the established data pipeline of query operations. Let's assume that table *Persons* contains a record Pavlos, who is 30 years old, and a record Natasha, who is 65 years old. Now Natasha opens a new account, so there is a new insertion in the data table *Account*. This will be picked up by the lower query operator that *scans* this table, and as there is no filter, it will be picked up and forwarded to its upper layer. There will be retrieved by the implementation of the *JOIN* operator. It will receive the record, join the latter with the corresponding record related to the *Natasha* data record, and push the processed record to the upper layer. Eventually, it will arrive at the driver, and the AI algorithm or the application code will receive it from the corresponding *iterator* that still remains open.

Now Pavlos opens a new account. As before, a new record will be added to the *Account* data table that will be evaluated by the query operator and pushed to the implementation of the *join*. The latter, however, will not find any data record of *Pavlos* to join the newly added account, as Pavlos is lesser than 60 years old, and hadn't been retrieved by the *filtering* operation. As a result, even if the newly added record was received, it won't reach the driver as it wasn't valid against the submitted query statement.

In this example we had two state transitions of the dataset, due to the two data modification operations that took place, the insertion of two new accounts. Our query was executed incrementally, thus continued to validate the dataset following the incremental transitions of state.

Going back to pilot#2 that drove the motivation for implementing the incremental analytics, the AI algorithm for risk assessment requires the calculation of the average value of the financial currency of a product per hour. This is translated to a query containing aggregation operations. As previously, it will be passed through the query compiler and eventually a query plan will be decided to be executed. This will eventually instantiate a data pipeline of query operations that formulate the tree of the query plan, and whose execution can be done incrementally. Instead of pilot#2 periodically send a query statement to retrieve *timely bounded* results, it can send the statement once, and each time a new data modifies the selected dataset, it will be evaluated and inform the driver. The AI algorithm now can respond/listen to

events receiving as the result of a data modification operation and can calculate the value at risk in real-time, thus avoiding opportunities that might have been occurred but missed during the time period between the two periodic invocations.

Under the scope of the task T5.2, we have extended the storage engine of the INFINISTORE to support the incremental analytics. The operations that take place in the storage layer are related to data access, and thus, are implemented at this layer. The data storage can identify incremental data transitions in a data table or data index. Moreover, it can be pushed down operations like filters. Therefore, at the second phase of the project, the INFINISTORE could initially support what we call *incremental scans*. This is the basic pillar for all other operations to be built upon, as we saw in our previous example. At the last phase of the project, we extended the incremental capabilities of all relational algebraic operations, except for the JOINS. This means that a data user or application developer can submit an aggregation query statement and continuously update the results in a visualization plot or feed this information to an AI algorithm. The latter can in fact consumer data coming from a stream and combine this information with the always updated aggregation value coming from the datastore. This is exactly what has been developed under the scope of the T3.3 (“Integrated Querying of Streaming Data and Data at Rest”). In combination with the outcomes of the work being carried out of the task T5.2 that is reported in this document, the data analyst can take advantage of both the integrated query processing, combining streaming data with the incremental analytics. To be able to do this, we have implemented their invocation from the direct API that the INFINISTORE exposes. This can be used in order to exploit such capabilities and has been integrated with the streaming operators developed in T3.3. The following subsections contain the documentation of such usage and an example on how to use this in practice.

3.4 Documentation

In this subsection, we will provide some details on how the direct API that allows the use of *incremental scans* has been designed. The class diagram of the main classes that should be used can be depicted in Figure 4.

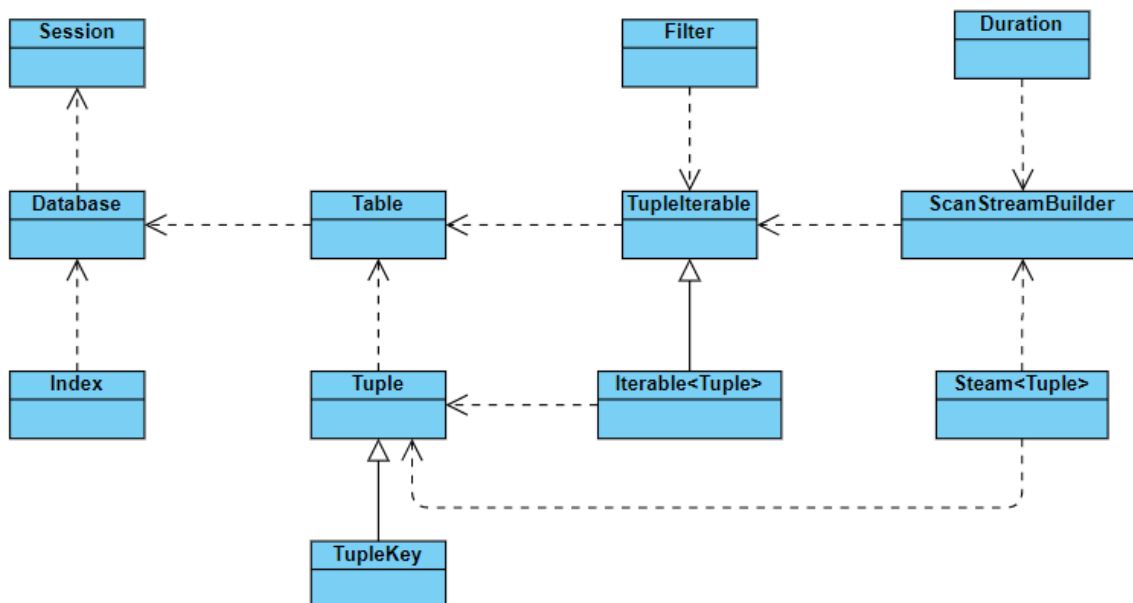


Figure 4: Class diagram of direct-API for incremental scans

The basic object that we need to first create is the *session*, which resembles the *Connection* in JDBC. It keeps a connection open with the storage layer of INFINISTORE, manages transactions and allows the application developer to perform operations on the connection level. From this object, we can connect to a specific *logical database* that we want to get or store data.

The object *database* allows us to do operations on the logical database level. Typical operations can be the creation of a data table, index or sequence, or the deletion of these structures. Another important category of provided methods is the one that return an instance to manage these structures. That way, the application developer can have access to a specific data table that he or she wants to perform some operations.

The object *table* provides access to a specific data table. The user can insert, update, delete or request a get or scan operation, in order to modify or retrieve data rows. In the storage engine, each data row is represented by an object of the class *Tuple*. This class can be further extended to *TupleKey*, *TupleUpdate*, *TupleDelete* etc, that provide some additional functionalities related to the corresponding operations. For instance, the *TupleKey* is a *Tuple*, where only the fields that consist of the primary key are fulfilled with values. One of the most important methods of the *Table* is the *filter* that allows the data user or application developer to perform a *scan* operation over a data table structure. As we saw, the result of such operation will always have to be an implementation of an *Iterator* that will allow the retrieval of data rows from the list of arrays that formulate the virtual table of the result of the *scan/filter* operation.

The *TupleIterable* is the result of the invocation of the *find* operation that allows the data user to retrieve data items. As it is depicted from the class diagram, it extends the *Iterable* generic interface, which is concretized by defining that it will iterate over objects of class *Tuple*. This is the class that holds information on the data row level. The *TupleIterable* can be declared to do a scan using a *Filter*. This resembles the WHERE clause in standard SQL, while there are additional methods to declare the projection of the result (equivalent to the SELECT clause of the standard SQL) etc. The data user or application developer can use this *TupleIterable* object to execute such statements in order to access the storage system and retrieve the data from the datastore.

All these classes and methods implement functionality for a static retrieval of data, or as we mentioned, allow the data users to perform operations over a *timely bounded* dataset. That is the snapshot of the living dataset at the specific point in time when the query (or the call to *scan*) occurs. So, how can we use our newly developed *incremental scans*? To do so, we need to create the *ScanStreamBuilder* object. This will eventually return a *Stream*, concretized by defining that this stream will be over *Tuples*. The *ScanStreamBuilder* provides methods to configure how the stream will behave: start, stop, being an infinite, or it has to receive steam for a specific *Duration* of time.

We will see all these in practice in the following subsection. The remaining of this one contains more detailed information of the classes defined in the class diagram of Figure 4, along with a documentation of the most important methods of each class.

Session

- `int getSessionId();` Returns unique identifier for this session.
- `Database database();` Provides access to Database object, which is the entry point for all the operations related to data.
- `void commit();` Commits all the operations done since the last begin transaction.
- `void rollback();` Rollbacks all the operations done since the last begin transaction.
- `boolean inTransaction();` Indicates whether the session has a transaction started or not.
- `void close();` Closes the current session. Every operation called after this call will throw an `IllegalStateException`.

Database

- Table `createTable(String name, List<Field> keyFields, List<Field> fields)`; Creates a new table in the database.
- Index `createIndex(String tableName, String indexName, List<Field> fields, boolean isUnique)`; Creates a new index on the table. The index name must not exist on the table, and cannot be null. The field list must not be empty.
- `boolean tableExists(String tableName)`; Return if a table exists or not.
- `Collection<Table> getTables()`; Retrieves the full list of database tables.
- `Table getTable(String name)`; Retrieves an existing table given its name.

Table

- `Tuple createTuple()`; Creates an empty tuple with the list of fields of the table format, including the primary key fields.
- `Tuple get(TupleKey key)`; Retrieves a single tuple from a given key.
- `Table upsert(Tuple tuple)`; Updates or inserts a tuple. If the tuple exists, is modified and if not, is inserted.
- `Table update(Tuple tuple)`; Updates an existing tuple. If the tuple doesn't exist (it means, a tuple with the same tupleKey) it will throw an Exception.
- `Table delete(TupleKey tupleKey)`; Deletes an existing tuple given its key. If the tuple doesn't exist it does nothing. If the tuple key is incomplete, it will scan the table searching for the tuples that match the part of the key that is defined, and it will remove all the tuples that match with the defined part of the tupleKey.
- `TupleIterable find()`; EntryPoint to run queries over the table. This method returns a `TupleIterable` object which allows to be configured and traversed to scan the table data.

TupleIterable

- `TupleIterable filter(Filter filter)`; Configures this `TupleIterable` to use a filter over the Table tuples.
- `TupleIterable first(long n)`; Configures this `TupleIterable` to return the first n tuples.
- `TupleIterable skip(long n)`; Configures this `TupleIterable` to skip the first n tuples.
- `TupleIterable project(Projection projection)`; Configures this filter to make a projection over the result tuple fields. If this method is called after an aggregation call, the projection will apply over the result aggregation fields, not the original table tuple fields.
- `TupleIterable sort()`; Configures this `TupleIterable` to return values ordered by the PK
- `TupleIterable reverse()`; Configures this `TupleIterable` to do the scan in a reverse order.
- `TupleIterable fromTupleTs(Tuple tuple)`; Configure the tuple to search from. This is, only tuples inserted or modified after the given tuple was inserted (or modified) will be taken into account by the scan. By default the given tuple (and all committed in the same commit) will not be included in the scan
- `TupleIterable fromNow(Session session)`; Configure the scan to search only tuples committed after this moment ts. This is, only tuples inserted or modified between the call of this method and the execution of the scan will be taken into account by the scan
- `ScanStreamBuilder asStream()`; Creates a stream builder that allows to get the scan results in the form of a stream.

ScanStreamBuilder

- `ScanStreamBuilder allowInternalCommits();` Allow commits from a stream lambda iteration. For example, if you want to go over the stream with a `foreach lambda` and inside that lambda you want to do some commit, you need to call this method, otherwise, the commits from the lambda will be rejected. On the other hand, if you allow internal commits you will not be able to see session modifications that are not committed.
- `ScanStreamBuilder infinite();` Configure the stream as infinite. After returning all the rows that matched with the scan at this moment will wait infinitely for new rows. The only way to stop the stream and return the execution control to the client is calling the method `stop`
- `ScanStreamBuilder duration(Duration duration);` Sets the max duration of this stream. The time starts to count when the stream method is called
- `Stream<Tuple> stream();` Creates and open a Stream to retrieve the tuples from the DB

3.5 Incremental scans in Practice

In this subsection we will provide a sample code snippet of how to make use of the *incremental scans* that have been developed during the second phase of the project. We will rely on a real user scenario taken from pilot#2 (“Real-time risk assessment in Investment Banking”) of INFINITECH. As we have mentioned, the scenario is that data related to the financial currencies of products are being ingested into the INFINISTORE, connecting the external stream to an interim Kafka queue that transparently stores the data stream to the datastore, making use of our developed *connector*, which has been implemented under the scope of the task T3.1 (“Framework for Seamless Data Management and HTAP”).

We will first setup the deployment and start sending data, and then we will write some code to get results incrementally, by using an infinite *Stream*. The *incremental scans* have been integrated with the storage engine of INFINISTORE and are now an integral part of it. Thus, we need to deploy the datastore. Having the INFINITECH way for deploying technological building blocks, we will make use of the specific blueprints and Kubernetes for the container orchestration. The following configuration will deploy the datastore.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: infinistore
  labels:
    app: infinistore
spec:
  serviceName: infinistore-service
  replicas: 1
  selector:
    matchLabels:
      app: infinistore
  updateStrategy:
    type: RollingUpdate
  podManagementPolicy: OrderedReady
  template:
    metadata:
      labels:
        app: infinistore
    spec:
      initContainers:
        - name: infinistore-home-fix
          image: busybox:1.30.1
          command: ["/bin/sh", "-c", "chown -R 999:999 /datasets"]
      volumeMounts:
```

D5.3 Library of Parallelized Incremental Analytics - III

```
- name: infinistore-datasets-storage
  mountPath: /datasets
containers:
- image: harbor.infinitech-h2020.eu/data-management/infinistore:latest
  name: infinistore
  ports:
    - containerPort: 2181
    - containerPort: 1529
    - containerPort: 9876
    - containerPort: 9992
    - containerPort: 14400
    - containerPort: 9800
  volumeMounts:
    - name: infinistore-datasets-storage
      mountPath: /datasets
  startupProbe:
    exec:
      command:
        - /bin/sh
        - -c
        - python3 /lx/LX-BIN/scripts/lxManageNode.py check QE
    timeoutSeconds: 5
    failureThreshold: 30
    periodSeconds: 10
  resources:
    limits:
      cpu: 4000m
      memory: 8Gi
    requests:
      cpu: 2000m
      memory: 4Gi
  env:
    - name: USEIP
      value: "yes"
    - name: KVPEXTERNALIP
      value: "infinistore-service!9800"
  restartPolicy: Always
  imagePullSecrets:
    - name: registrysecret
  volumes:
    - name: infinistore-datasets-storage
      persistentVolumeClaim:
        claimName: infinistore-datasets-pvc
```

We would also need to define a *service* to expose the various ports to access the storage engine:

```
apiVersion: v1
kind: Service
metadata:
  name: infinistore-service
  labels:
    app: infinistore
spec:
  ports:
    - name: "9876"
      port: 9876
      targetPort: 9876
    - name: "9992"
      port: 9992
      targetPort: 9992
    - name: "14400"
      port: 14400
```

D5.3 Library of Parallelized Incremental Analytics - III

```
targetPort: 14400
- name: "9800"
  port: 9800
  targetPort: 9800 selector:
app: infinistore
```

Then, we will deploy the kafka queue that will connect to the INFINISTORE. We will also make use of the corresponding blueprint:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: lx-kafka
  labels:
    app: lx-kafka
spec:
  serviceName: lx-kafka-service
  replicas: 1
  selector:
    matchLabels:
      app: lx-kafka
  updateStrategy:
    type: RollingUpdate
  podManagementPolicy: OrderedReady
  template:
    metadata:
      labels:
        app: lx-kafka
    spec:
      containers:
        - image: harbor.infinitelab.com/interface/lx-kafka:latest
          name: lx-kafka
          ports:
            - containerPort: 8081
            - containerPort: 9092
          resources:
            limits:
              cpu: 2000m
              memory: 2Gi
            requests:
              cpu: 1000m
              memory: 1Gi
          env:
            - name: advertised_url
              valueFrom:
                configMapKeyRef:
                  name: lx-kafka-configmap
                  key: advertised.url
            - name: advertised_port
              valueFrom:
                configMapKeyRef:
                  name: lx-kafka-configmap
                  key: advertised.port
            - name: topics_total
              valueFrom:
                configMapKeyRef:
                  name: lx-kafka-configmap
                  key: topics.total
            - name: connection_url_1
              valueFrom:
                configMapKeyRef:
                  name: lx-kafka-configmap
```

D5.3 Library of Parallelized Incremental Analytics - III

```
      key: connection.url.1
- name: topics_1
  valueFrom:
    configMapKeyRef:
      name: lx-kafka-configmap
      key: topics.1
- name: connection_database_1
  valueFrom:
    configMapKeyRef:
      name: lx-kafka-configmap
      key: connection.database.1
- name: database_tablename_1
  valueFrom:
    configMapKeyRef:
      name: lx-kafka-configmap
      key: database.tablename.1
- name: pk_fields_1
  valueFrom:
    configMapKeyRef:
      name: lx-kafka-configmap
      key: pk.fields.1
- name: fields_whitelist_1
  valueFrom:
    configMapKeyRef:
      name: lx-kafka-configmap
      key: fields.whitelist.1
restartPolicy: Always
imagePullSecrets:
  - name: registrysecret
```

The kafka queue needs a *ConfigMap* to configure the topic to listen to, the target data table, its primary key and fields. For the scenario of pilot#2, the configuration should be as follows:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: lx-kafka-configmap
data:
  advertised.url: rancher.vps.uninova.pt
  advertised.port: "30202"
  topics.total: "1"
  connection.url.1: infinistore-service
  topics.1: tickdata
  connection.database.1: JRC
  database.tablename.1: TickData
  pk.fields.1: PRODUCT, DATETIME
  fields.whitelist.1: TIK_OPEN, TIK_HIGH, TIK_LOW, TIK_CLOSE, TIK_UP, TIK_DOWN
```

Here the name of the *logical database* is 'JRC', while the data table to store the data stream is 'TickData'. We will use those names in our code of the *incremental scans*.

To start a data stream so that data can be loaded, we can make use of the stream simulator we have developed for the needs of this pilot. It can be found at the project's repository³. Now that we have everything in place, we start the simulator, and we can see data being generated and send to the queue, and from there, down to the datastore.

³ https://gitlab.infinitech-h2020.eu/pilot_2/ticksimulator

The following code snippet demonstrates how we can make use of our *incremental scans*:

```
Settings settings = Settings.parse("lx://lxserver:9876/JRC@APP");

try(SessionImpl session = (SessionImpl) SessionFactory.newSession(settings)){
    Table tickData = session.database().getTable("TickData");

    //waits for all new tuples committed to the DB from this moment and return it in the stream
    tickData.find().fromNow(session).asStream().allowInternalCommits().infinite().stream().forEach(tickDataRow ->{
        //do something with the row
        System.out.println("new row: " + tickDataRow.toString());
        session.commit();
    });
}
```

In this sample code, we first open a connection using the *SessionFactory* to create the *Session* object. It makes use of the *Settings* object that is configured to connect to the specified location. Then, based on the documentation of the previous subsection, we create the *Database* object, that will connect to the *JRC* database, as defined in the connection URL, and then we will get the *Table* object to access the *TickData* table. Then, we need to execute a *scan* operation. We invoke the *find* method that returns the *Iterable* object. We did not provide any input argument to this method, so this is going to do a full scan over the *TickData* table. However, as we need to get a *Stream* of *Tuples* and get results as they are being added, we invoke the *asStream* method that returns the *ScanStreamBuilder* object. At this point, we need to configure our stream, before executing. We define that this will be an *infinite* stream and that we are allowed to do internal commits using the same *session* object we had for opening the stream. Finally, the *stream* method opens the stream, returns an object of the class *Stream<Tuple>* on which we can iterate using lambda expressions. In our sample code, we print the data item to the output of the console.

As we can see from our code snippet, even if the concepts behind the design and implementation of the *incremental scans* might seem complex, the exposed interface is fairly simple and intuitive. We manage the *incremental scans* the same way we manage normal scans, with the difference of having to configure the *ScanStreamBuilder*. Concerning best practices and design patterns, the use of the *Builder* makes it easy for the application developer to configure the stream and get the newly added records that satisfy the *filter conditions* in a Java native way, as he or she can iterate as with any other type of object.

As we have mentioned, during the last phase of this task, we have extended the INFINISTORE incremental capabilities for all relational algebraic operations, except for JOIN. This means that we can submit any type of aggregation operator and get the updated results from the stream builder. The reason for excluding the JOIN operator is that it does not give any benefit. When a new record is being pushed from the storage to the upper layers of the tree of the query execution plan, when it reaches the JOIN operator, it will need to be sent further down on the right part of the JOIN. This creates a significant performance overhead that will out scope the benefits of the incremental query. Moreover, there was no real case scenario that could benefit from such operators and therefore, it was decided to give focus and spend effort on the implementation of the *source* part of the Kafka connector, that will be described in the next section.

4 Kafka source connector

Having the capabilities for incremental analytics ready and having extended the INFINISTORE direct API to support this functionality, it was envisioned that this could be beneficial for the *Change Data Capture* (CDC) paradigm. Moreover, the semantic interoperability engine of INFINITECH that is being developed under the scope of the activities of WP4 (“Interoperable Data Exchange and Semantic Interoperability”) could make use of this, in order to have a continuous snapshot of the data being ingested to the datastore that need to be transformed into their internal tripe-store. This communications requires the use of intermediate data queues. In INFINITECH, we rely for these purposes on the Kafka data broker. The latter exposes an interface that can be implemented by external data *connectors* in order to get access to a data management layer. Under the scope of T3.1 (“Framework for Seamless Data Management and HTAP”), we have implemented the *sink* part of the INFINISTORE connector. That allows the ingestion of data from Kafka to the datastore. Under the scope of the T5.2 that is reported in this document, we have additionally implemented the *source* part of our connector. In the next subsection we provide details of this implementation, along with a demonstrator of its use.

4.1 Core implementation of the source connector

Kafka exchanges data from or to an external data management system as a series of data records. Each data record that is being sent or retrieved to the datastore is mapped under the *SourceRecord* class. This defines an array of Objects, which resembles the columns of a data row. Moreover, it defines the *SourceTask* abstract class that is used by the Kafka engine to retrieve data from an external datastore. Therefore the provider needs to extend this class and write the source code to allow this data retrieval. Figure 5 shows the implementation for the INFINISTORE.

The *SourceTask* has been extended by the *LXSourceTask*. This implements the logic for retrieving data from the INFINISTORE. However, as the data could be ingested continuously, there must be a different thread that takes the responsibility of reading data from the datastore in a continuous manner and send them to the *LXSourceTask*, which is our implementation for the *SourceTask* that has been started by the Kafka core engine. This thread makes use of an instance of the *LXReader* class.

The purpose of the *LXReader* class is to connect to the INFINISTORE, read data and send them to the listening object used by Kafka. It will open a *KiviSession*, which is a connection to the datastore and submit the query statement. In a simple scenario, that might be a full scan of a data table that will allows the retrieval of data coming from the datable in a continuous manner. Taking advantage of the incremental analytics described in the previous subsection, we can submit an *unbound* infinite query that will create an *iterator* to get all data records stored in the target data table. Once the full scan ends, the iterator will remain open and wait for new data to become available. When a new record is stored into the storage engine of the INFINISTORE, it will be propagated via its direct API to the iterator that has remained opened, waiting for new records. The new ingested record then will be pushed to the *LXSourceTask* that is being used by the kafka core engine itself, and eventually will be sent to the specific kafka topic to become available to any possible listeners or consumers of such data.

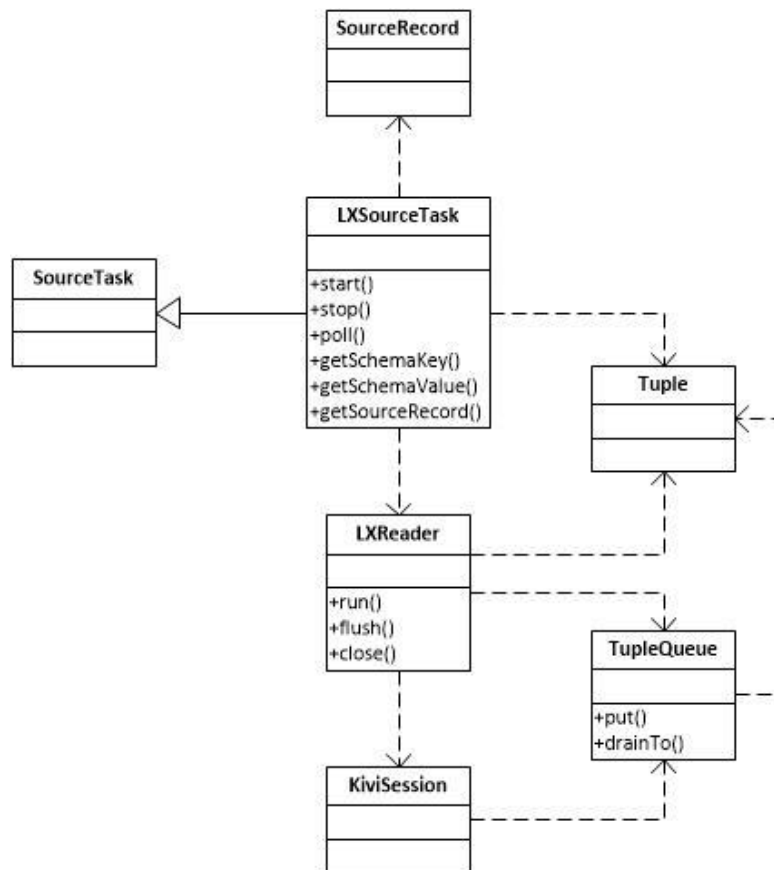


Figure 5: Class diagram of the Kafka source connector

Let us now see how this works: The *LXSourceTask* is instantiating the *LXReader* in a separate thread, sending all connection parameters for the latter to open the specific *KIVISession* and establish a connection to the datastore. Moreover, it creates a *BlockingQueue* to exchange data coming from the datastore between the *task* and the *reader*. This is the purpose of the *TupleQueue*. Data that are read from the incremental operator coming from the *KiviSession* are being sent to this queue which is periodically asked by the *LXSourceTask* to pop up the data. Data retrieved from the datastore are stored into rows that are wrapped in a *Tuple*. Therefore, the *KiviSession* sends *Tuples*, that are being put to the *TupleQueue*, which in fact is being asked from the *LXSourceTask* to periodically retrieve them. Then, the *LXSourceTask* transforms the *Tuple* into a *SourceRecord*, using the record's schema, as defined by the data user. The *SourceRecord* is serialized then using the *AvroRegistry* based on this schema, and is being placed into the *Kafka topic* by the Kafka core engine itself. The following list contains the core methods that have been developed in our implementation.

LXSourceTask

- **start:** this an abstract method defined in the *SourceTask* that is being called by the kafka core engine, once the *task* to retrieve data from our external datastore is being started. In this method, we create an instance of the *LXReader* that will be started in separate thread, and initializes all internal attributes.
- **poll:** after the instantiation of the source task, the kafka core engine periodically invokes this method to retrieve a list of *SourceRecords*, which actually contains the data being retrieved from the datastore. Internally, it will pop records from the *TupleQueue* in an array of *Tuples*, and it will transform them to the *SourceRecord* that Kafka uses.
- **stop:** once the task finishes or gracefully closes, the Kafka core engine invokes this method, whose implementation makes sure to properly shutdown the thread of the *LXReader*, as well as releasing all other open resources.

LXReader

- **run:** the *LXReader* implements the *Runnable* interface, and therefore implements this method. Internally, this method opens a *KiviSession*, creates an *Iterable* object that has been constructed accordingly, and retrieves data from the INFINISTORE. When data are retrieved, they are being put to the *TupleQueue*. The iterator remains open, until the invocation of the *close* method takes place.
- **close:** the *LXReader* also implements the *Closeable* interface, and this method is invoked by the *LXSourceTask* when its *stop* method is called by the kafka core engine. It terminates the execution of the *iterator* and closes the *KiviSession*, releasing all its open resources.
- **flush:** this method is being invoked by the *LXSourceTask*, when its *poll* method is called by the kafka core engine. It invokes the *drainTo* method of the *TupleQueue*, which is not blocking, and returns a list of *Tuples* to be further processed by the *LXSourceTask*.

TupleQueue

- **put:** this a method implemented by the *BlockingQueue* Java implementation, and is called by the *LXReader* to send data to the queue. It blocks if the number of tuples in the queue is bigger than its size.
- **drainTo:** this a method implemented by the *BlockingQueue* Java implementation, and is called by the *LXReader* to get data from the queue. It is not blocking and returns a list of the *Tuples* that are currently stored in the queue, while also removes those tuples from the queue.

4.2 Kafka source using filters

The main purpose of a source connector is the retrieve data coming from an external source and put them into a Kafka topic. This means that it will wait for new data to be available, and when this happens, they are being retrieved by the kafka core engine using the custom implementation of the connector. In our case, this makes use of the INFINISTORE incremental capabilities using its direct API. As it has been described in the previous subsection, the INFINISTORE allows for incremental scans, while additionally enhanced all algebraic operations (except for JOINS) with these incremental capabilities. Having that, we progress a step beyond the current state-of-the-art, allowing the data user to define some filter criteria, which will be transformed to a filter condition. The latter will allow the retrieval of only the data that arrives into that storage engine of INFINISTORE and satisfy the filter condition.

The data user defines these filter criteria in a similar way as in the WHERE clause of a relational SQL query statement. For instance, the following is valid:

```
age>60 AND (name like 'Constantine' OR name startswith 'P')
```

This text needs to be firstly compiled into a structured manner that can be used by our *LXReader* to create the appropriate *Filter* condition, when establishing the data connection with the INFINISTORE. Therefore, it will be firstly parsed and transformed to a *FilterElement*. Then the latter will be translated to an instance of the *Filter* class, used by the direct API of INFINISTORE to submit the query statement. Figure 6 illustrates the class diagram used for this purposes.

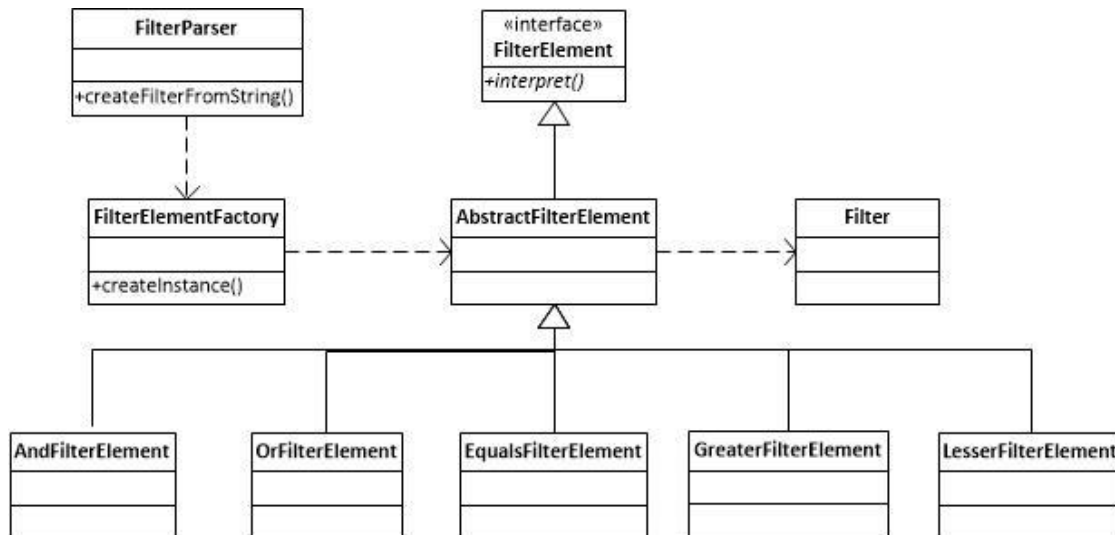


Figure 6: Class diagram of filter conditions in kafka

As it can be depicted, the *FilterParser* accepts a String object, which has the filter criteria, and after compiles it using the *FilterElementFactory*. The latter, constructs on *FilterElement* for each condition that is included in the input string with the filter criteria. This is an interface that defines the *interpret* method. It is implemented by the *AbstractFilterElement* which, as its name proves, it is an abstract class. Then, there is an extension of this abstract class for each of the filter operators that are supported by the INFINISTORE internal query engine. Therefore there is one class for the “AND” operator, one for the “NOT” operator, another for the OR operator, as for the EQUALS (“=”), GREATER (“>”), lesser (“<”), GREATER_OR_EQUALS (“=>”), as long as for String operators like CONTAINS, LIKE, STARTS, etc. It is important to mention that in that figure, we have put a non exhaustive list. As its operator has its own class, each of the class implements the *interpret* method accordingly. The result will be an instance of the *Filter*.

4.3 Kafka source using aggregations

In the same manner as we did for the filter conditions that can be pushed down to the INFINISTORE, having the incremental analytics developed under the scope of this task, we can now take benefit from the incremental aggregation operators. This will allow to retrieve aggregated information from the datastore in an incremental manner that can be further consumed by a software building block or AI algorithm that makes use of a Kafka sinker. Figure 7 illustrates the class diagram of our implementation.

In a similar way with the filter conditions, the data user can define an aggregation in the configuration file of our connector. This definition will be used by the *AggregationParser* to create instances of *AggregationElements*. The latter is an abstract class that is extended by each of the 5 aggregation operators that can be found in the relational algebra: min, max, count, sum, avg. All these classes implement the *toAggregation* method that transform their input into the specific *Aggregation* operator that will be used by the *Iterable* object of the *KiviSession* that will establish the connection to the INFINISTORE. We support filters, aggregations, and aggregations have the GROUP BY clause, and filters over the result of such aggregations, which resembles to the HAVING clause of the relation SQL language.

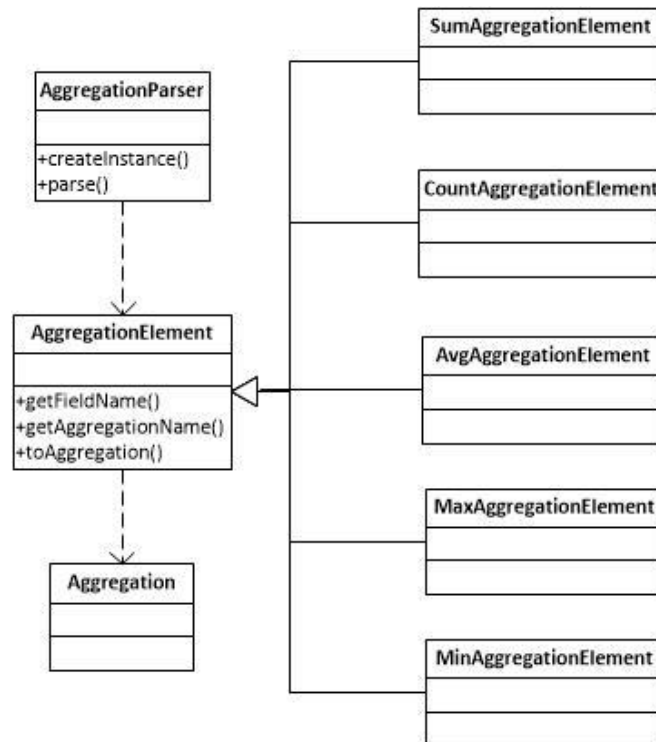


Figure 7: Class diagram of aggregations in kafka

Concluding, apart from only implementing the incremental analytics as an enabler for advanced analytics provided by the INFINISTORE, this task went beyond its original scope and we have additionally implemented the Kafka source connector that makes use of such capabilities. Moreover, we defined a SQL alike language so that the data user can execute queries, using relational algebraic operators that will be submitted against a specific table, and the results can be retrieved in a continuous manner via a Kafka broker. The following subsection demonstrates the use.

4.4 Kafka source connector in practice

In order to demonstrate our implementation, we will need the deployment of INFINISTORE, along with a Kafka broker. We have already demonstrated how to deploy and interconnect these components in practice, in the previous subsection. The reader can make use of the same manifest scripts to conduct Kubernetes to start the containers. He or she will only need to define the properties to be used by the kafka source connector. The following code snippet shows an example of its use.

```

name=local-lx-source
connector.class=com.leanxcale.connector.kafka.LXSourceConnector
tasks.max=1
topics=mytopic
key.converter=io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url=http://localhost:8081
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://localhost:8081

connection.properties=lx://localhost:9876/test@APP

source.table=LX_SOURCE_TABLE
source.queue.size=11110
source.from.mode=offset
source.topic.name=${table}
source.fields.whitelist=
source.pk.fields=
  
```

D5.3 Library of Parallelized Incremental Analytics - III

```
source.filter.datetime.pattern=yyyy-MM-dd HH:mm:ss
source.filter=FIELD1=VALUE2 AND NOT (FIELD2 STARTS 'This is a string' OR FIELD3='2004-03-21
00:00:00')
source.aggregations=count, avg:field3, sum:field3
source.aggregations.groupby=field2
source.aggregations.having=field2>value
```

During run-time, the connector is able to connect to a INFINISTORE table (specified by the property "source.table") and perform as a streaming pipe to a Kafka Topic. The connector will automatically register the schema on the schema registry and create the topic according to that schema. By default, all table fields will be mapped, but you can just map (and copy) some of them using the property "source.fields.whitelist". Other important property is "source.from.mode". With this property you can manage the tuples to be sent to kafka. You can copy all tuples (begin mode), new tuples from last run, stored in kafka offset property, (offset mode) or just tuples that are stored, or modified, since the connector is running (now mode). In case you define a list of fields in the "source.pk.fields" property, then these fields will be considered as the primary key of the tuple that is being sent to the Kafka Topic. The user can also define a filter condition to be applied to the INFINISTORE table in order for the streaming pipe to send only the tuples that validates this filter to the Kafka Topic. The filter can be defined in the "source.filter" attribute, while the user can also define the string pattern for a datetime value of the filter, in the "source.filter.datetime.pattern" attribute.

The source connector also supports the definition of aggregation incremental operations. This can be done by adding the following property: "source.aggregations". An indicative example about the format that this property expects is the following: "count, avg:field3, sum:field3". If the user wants to add a 'group by' operation that will be used for calculating the aggregations, this can be done by adding the list of the 'group by' fields in the property: "#source.aggregations.groupby". Finally, the user can filter the results of the aggregations, similar to the 'having' clause of the SQL, by adding the property: "source.aggregations.having". This property expects a filter condition, as explained previously.

5 Conclusions

This report documented the work that has been carried out in the scope of task T5.2 “Incremental and Parallel Data Analytics” until this final phase of the project. The main objective of this task is on one hand to deliver a set of algorithms that are currently being used frequently in the insurance and finance section, relying on well-known families of machine learning implementations for correlation discovery and forecasting based on time-series prediction, clustering and collaborative filtering. The goal is to make use of these algorithms, parallelize them and make them incremental. This is enabled by accomplishing the second objective of this task which is the provision of incremental analytics. During the second and third phase of this task, we focused on extending the storage engine of the INFINISTORE in order to allow the propagation of data modifications that can be further used at the level of the query engine. The latter can now be configured via a well-defined API to return the results of a submitted query incrementally. An additional effort was given, extending the work that was initially planned to be carried out by this task, to implement the Kafka source connector that makes use of the incremental analytics of INFINISTORE.

This deliverable firstly gives an overview of how we plan to parallelize algorithms for collaborative discovery and time-series prediction. Secondly, we provided the motivation behind the implementation of *incremental analytics* and the basic principles and design of our solution. Furthermore, we added documentation regarding the use of the *incremental analytics*, along with some examples using code snippets that rely on a rear user scenario taking from pilot#2 of INFINITECH. During the last phase (M20-M27) we have also implemented a Kafka source connector that makes use of our *incremental analytics* to send data modifications to other target data sources. This will be exploited by the Intelligent Data Pipelines of INFINITECH, defined and implemented under the scope of T3.4. This will enable the integration of the Semantic Interoperability Engine of the platform with its data management layer. It is important to highlight here that we went beyond the state-of-the-art in such type of connectors to external data sources using Kafka brokers by allowing the user to define SQL relational algebraic operations in the configuration file, that will be further transformed and pushed down to the storage engine of INFINISTORE, thus enabling the execution of SQL relational statements, in a continuous and incremental manner, whose results can be retrieved by Kafka data queues.

The technological outcomes of the work that has been carried out under the scope of this task have a significant impact on LeanXcale. They have been already incorporated into LeanXcale’s main product, the LeanXcale datastore, which allows for the provision of incremental analytics. This was crucial as it allowed for LeanXcale to investigate new opportunities targeting markets that makes use of data intensive pipelines. Currently, there are several well established Proof-of-Concepts with potential clients. Additionally, the Kafka source connector implemented in this task is part of the supportive products of LeanXcale. Having this, it allows for its clients to explore different ways of communicating with the datastore, thus allowing for a plethora of different solutions and making it more feasible for LeanXcale to interact with the organizations’ internal frameworks and mainframes.

Table 1: Conclusions (TASK Objectives with Deliverable achievements)

| Objectives | Comment |
|---|---|
| <i>parallelize popular incremental analytics algorithms</i> | This task provided the enabler for popular analytical algorithms to consume aggregated and pre-processed data in an incremental manner. There has been developed the incremental analytics that are provided by the INFINISTORE data management system, allowing data users to submit continuous and un-bounded queries down to the storage engine. The scope was not much given in parallelize some specific algorithms, rather than provide incremental capabilities for all types of algorithms to make use of, pushing down this functionality to |

| | |
|---|--|
| | the data management layer, and thus allowing the algorithms to be implemented in a parallel way. The provision of such algorithms are the scope of T5.4, while the parallelization takes place in combination with the outcomes of this task and T3.4. |
| <i>exploit of accurate local knowledge that will enable optimization in the aggregating nodes each time a snapshot update is needed.</i> | This task implemented the incremental analytics that go beyond the scope of this objective. Instead of providing local knowledge that can enable the optimization each time there is a need for a snapshot of data, the incremental analytics always provide the current status, thus updating the result of aggregation operators the moment data are being ingested. Based on the transactional database capabilities of the INFINISTORE, this ensures data consistency and thus accuracy of results. There is no need for optimization in the aggregation nodes of popular algorithms, as these nodes can be completely removed and let the INFINISTORE do the actual work using its incremental analytics. |
| <i>use of approximations, including relevant trade-offs between performance and accuracy of the results, in the cases of complex incremental algorithms</i> | Again, by implementing the incremental analytics in the data management layer, there is no need for approximations and trade-offs between performance and accuracy of the results of an AI algorithm, as this is guaranteed by the data management layer, and it will be always accurate, as the INFINISTORE provides data consistency using database transactions. AI algorithms can now push down this functionality to the datastore that supports aggregated operations to be submitted in an incremental manner. |

Table 2: Conclusions – (map TASK KPI with Deliverable achievements)

| KPI | Comment |
|---|--|
| <i>Parallel and Incremental Analytics Framework</i> | <i>Target Value</i> >= 1 The specific KPI is successfully achieved with the presented solution, the incremental analytics provided by the INFINISTORE data management layer. |
| <i>Average reduction in latency of ML/DL algorithms execution</i> | <i>Target Value</i> >=30% The specific KPI has been partially achieved by the work that has been carried out under the scope of the T5.2. With the advancements of this work, namely the framework for incremental analytics along with the provision of the corresponding Kafka connector, we have completely removed the latency needed for data preprocessing from the data management layer. Now, the latter can allow the submission of continuous non-bounded query and provide the results in real-time, as data is being further ingested to the datastore, without the need for re-submitting a query that would have created a latency between the time of the submission and when the results have been retrieved. |
| <i>Reduction of Effort for development of ML/DL functionalities in the Sector</i> | <i>Target Value</i> >=50% The specific KPI has been partially achieved by the work that has been carried out under the scope of the T5.3. With the advancements of this work, namely the framework for incremental analytics along with the provision of the corresponding Kafka connector, there is now no need for the developer of ML/DL functionalities to implement data |

| | |
|--|--|
| | parallel and incremental data processing data nodes, as these can be completely delegated to the INFINISTORE, which now provides the capabilities for incremental analytical processing. |
|--|--|

6 References

- [1] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in timeseries databases. In Proceedings of the International Conference on Management of Data (SIGMOD), pages 419-429, 1994.
- [2] R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In Proceedings of the International Conference on Foundations of Data Organization and Algorithms (FODO), pages 69-84. Springer-Verlag, 1993.
- [3] A. Mueen, Y. Zhu, M. Yeh, K. Kamgar, K. Viswanathan, C. Gupta, and E. Keogh. The fastest similarity search algorithm for time series subsequences under euclidean distance, August 2017. <http://www.cs.unm.edu/~mueen/FastestSimilaritySearch.html>.
- [4] K. Chan and A. W. Fu. Efficient time series matching by wavelets. In Proceedings of the International Conference on Data Engineering (ICDE), pages 126-133. IEEE Computer Society, 1999.
- [5] E. J. Keogh, K. Chakrabarti, M. J. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. Knowledge and Information Systems (KAIS), 3(3):263-286, 2001.
- [6] R. Cole, D. Shasha, and X. Zhao. Fast window correlations over uncooperative time series. In Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD), pages 743-749. ACM, 2005.
- [7] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. J. Keogh. Beyond one billion time series: indexing and mining very large time series collections with iSAX2+. Knowledge and Information Systems (KAIS), 39(1):123-151, 2014.
- [8] Yagoubi, D.-E., Akbarinia, R., Kolev, B., Levchenko, O., Masegla, F., Valduriez, P., Shasha, D., 2018. ParCorr: Efficient Parallel Methods to Identify Similar Time Series Pairs across Sliding Windows. Data Mining and Knowledge Discovery, vol. 32(5), pp 1481-1507. Springer.
- [9] R. J. Hyndman and G. Athanasopoulos. Forecasting: Principles and Practice. <https://otexts.com/fpp2>
- [10] Brown, R. G. (1959). Statistical forecasting for inventory control. McGraw/Hill.