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

D5.6 – Framework for Declarative and Configurable Analytics - III

Revision Number	3.0
Task Reference	T5.3
Lead Beneficiary	LXS
Responsible	Ricardo Jiménez-Peris
Partners	LXS, ctag
Deliverable Type	Report (R)
Dissemination Level	Public (PU)
Due Date	2022-03-31
Delivered Date	2022-04-05
Internal Reviewers	JSI, NOVA
Quality Assurance	CCA
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, Jacob Roldan, Rogelio Rodriguez, José María Zaragoza Alejandro Ramiro Pavlos Kranas
CTAG	Contributor	Andrea Becerra García
NOVA	Internal Reviewer	Guilherme Brito
JSI	Internal Reviewer	Maja Skrjanc
CCA	Quality Assurance	Dimitris Drakoulis

Revision History

Version	Date	Partner(s)	Description
0.1	2022-03-01	LXS	ToC Version
0.2	2022-03-29	All	Addition of section 6
0.3	2022-03-29	LXS	Update the intro and conclusions
0.4	2022-03-29	LXS	Finalize the document
1.0	2022-03-29	LXS	Submitted for internal review
1.1	2022-03-30	NOVA	Internal review
1.2	2022-04-04	JSI	Internal review
2.0	2022-04-04	LXS	Submitted for internal QA
3.0	2022-04-05	LXS, GFT	Version ready for the submission

[1] Lead Beneficiary, Contributor, Internal Reviewer, Quality Assurance

[2] Can be left void

Executive Summary

The goal of Task T5.3 "Declarative Real-Time Data Analytics" is to provide a framework that can be considered as the enabler for real-time (online) data analytics over the data management layer of INFINITECH, in a declarative way, hiding the complexities of the implementation on the framework itself. Typical scenarios in both finance and insurance sectors require the continuous ingestion of data coming from a variety of sources at high rates, while at the same time they require the ability to perform complex and time-demanding analytical operations over the operational dataset, as the data is being inserted. Due to the current technological obstacles of combining those two different types of workloads, complex architectures are usually proposed and integrated into the current enterprise applications that allow performing such analytical operations. However, these analytical operations are performed over a snapshot of a database that is outdated, and not on real-time data. Those architectures are on one hand hard to implement, and additionally very difficult to maintain due to their complexity.

The INFINITECH project aims to provide an innovative data management platform that overcomes the current technological barriers throughout the data lifecycle, from data collection and ingestion, to analytical processing. The outcomes of this task is part of the *Advanced Analytical Processing* layer of the platform and are built on top of the fundamental pillars implemented in the work that has been carried out in the corresponding tasks of WP3, which provides the enablers for these advanced analytics. As a result, *the Declarative Real-Time Analytics* supports the declaration, configuration and execution of analytical queries over an operational datastore, and is able to return the result of those analytics in real-time, or as we better call it, *online*. This is a key requirement for the acceleration of the parallelized algorithms and additionally benefits the correlation of data *at-rest* with data *in-flight*, used in the streaming processing framework of INFINITECH. As the results can be retrieved online, they are acquired with minimum latency as opposed to traditional executions of analytic operations, and the results are always up-to-date with the operational data. We have extended the vanilla data structure of our data schema with the addition of new analytical columns that aim to facilitate the query execution, thus facilitating the analytical algorithms to retrieve post-processing results by directly connecting to the operational data store, instead of using complex and hard to maintain alternative architectures. Therefore, we have introduced the term *online aggregates* to refer to data analytics over real-time data, where the requirement is to retrieve results with the minimum latency possible, while ensuring data consistency at the same time. The execution of those *online aggregates* has been designed to be declarative, as they can be defined and configured via SQL DDL statements and their execution relies on the standard SQL syntax.

This deliverable introduces the *online aggregates* and describes the general concepts and design principles behind their implementation. It provides examples explaining the problem statement and the motivation behind them. We firstly provided the initial design of this framework, which is based on the semantic multi-version concurrency control implementation of the central data repository of the project. The design of this prototype along with an initial implementation had been already delivered in the first phase of the project, integrating the core mechanism of this component with the query engine of the central data repository, whose improvements and features have been already described in the corresponding deliverables of T3.1 and T3.2. This integration allowed the support of the definition of the *online aggregates* in a declarative way, using standard SQL syntax and delegating their execution to the query engine of the data management layer. As a result, significant effort had been put into the design of the SQL extensions and the documentation on how to use the real-time analytics. This documentation has been provided in deliverable, to allow the developers of the analytical algorithms of the pilot cases to utilise them. At the end of the second phase of the project, the integration with the query engine had been delivered, and thus, more advanced features and a combination of aggregate operations can be now supported and are described in the corresponding additional section of this deliverable. We then gave a more technical information regarding the execution of such statements by the query, along with initial benchmarking results, comparing our implementation against the traditional use of such statements. In this third and final version of this document, we additionally provide the results of an extended benchmarking that took place

in order to validate the performance evaluation of our implementation regarding the overhead that is expected to be added during data ingestion at very high rates.

Table of Contents

List of Figures

List of Tables

Abbreviations/Acronyms

2PL	Two-phase Commit protocol
AI	Artificial Intelligence
API	Application Programming Interface
BI	Business Intelligence
CSV	Comma-separated values
CTS	Commit Timestamp
DDL	Data Definition Language
ETL	Extract, Transform, Load
HTAP	Hybrid Transactional and Analytical Processing
IoT	Internet of Things
JDBC	Java DataBase Connection
KPI	Key Performance Indicator
NoSQL	No/Not only SQL
OLAP	Online Analytical Processing
OLTP	Online Transactional Processing
SQL	Structured Query Language
WP	Work Package

1 Introduction

The term real-time analytics refers to the ability of the framework or the architecture to provide analytical processing capabilities over the live-data, and not over a snapshot of the dataset that has been taken either by a periodically executed process (i.e. using Extract/Transform/Load - ETLs) or via the data migration from the operational datastore (where the real data lives) to a data lake or an analytical data warehouse via micro batching using intermediate data queues.

Typically, real-time analytics are very hard to achieve as a major requirement is the ability to ingest data at very high rates, while at the same time, compute large aggregate queries over the real-time data.

This scenario is often a common use case in BigData applications where data is being ingested at high rates and the analytical algorithms need to compute KPIs or other metrics over the real or live data, while it continues to be ingested. Such use cases can be found in the area of performance monitoring, in cases where there is the need to calculate aggregated values of IoT sensors, e-Advertisements, smart grids, industry 4.0, online detection of opportunities of risk assessment or other applications in the finance sector like online fraud or anti-money laundering detection, etc.

As data need to be stored in a persistent medium, data store solutions coming both from the SQL or NoSQL ecosystems seem to be inadequate to perform such operations.

In particular, SQL databases find this kind of workload troublesome, as they are not efficient at ingesting data at very high rates, and in the meantime, aggregate analytical queries are very expensive, as they require to traverse very large amounts of data, and in many cases, the whole data table, very frequently. This, in combination with the fact that traditional approaches rely on the 2PL (two-phase locking) mechanism for ensuring transactional consistency, makes it impossible for them to execute analytical query operations on the operational and real-time data.

On the other hand, NoSQL databases can handle the data ingestion at high rates, thus being effective on serving this type of workload. However, they cannot be used to execute analytical queries, and in many cases they rely on external analytical frameworks to provide this type of functionality to the data user. They cannot be used in scenarios where operational guarantees in terms of transactional semantics are required, and they lack support of transactions.

As a result, modern approaches in the industry rely on solutions that need to combine different data management technologies to solve these kind of use cases. This yields complex architectures that are very hard to be implemented successfully, and in any case, they are also very expensive to maintain.

The INFINITECH platform differentiates from those approaches by providing a framework for online data analytics, in a declarative manner, which is introduced in this report. We call this differential feature as *online aggregates,* a new technology that is being developed in the scope of this task, which is based on the brand new semantic multi-version concurrency control mechanism and enables the computation of aggregates (i.e. max, min, count, sum and average) in an incremental and real-time manner, using additional data structures in the storage engine. This creates new analytical columns that can be used by analytical algorithms. The latter can be facilitated by seamlessly using those columns while directly connecting to the data management layer of the platform.

The specific benefit that *online aggregates* offer is to enable the update of the relevant aggregate tables as data is being ingested. As so, aggregates are always pre-computed and the values can be retrieved online without the need to traverse the whole data table, an operation that has significant implications on the latency. Thus, a typically large and expensive analytical aggregate query becomes an almost costless query that reads one or few rows of those introduced aggregate tables that contain the analytical columns. With our approach, data ingestion becomes slightly more expensive in terms of latency, however, the implementation of the Online Transactional Processing (OLTP) engine of the INFINITECH data management layer, as described in D3.1 ("Hybrid Transactional/Analytics Processing for Finance and Insurance

Applications – I”), enables the sufficient execution and can handle very high throughputs. The slightly more expensive data ingestion removes the cost of computing the aggregates in real-time.

In this report, we introduce the concept of *online aggregates*, showcasing how both a traditional SQL database and a NoSQL datastore fail to solve the problem, and we explain how this new technology developed in the scope of the INFINITECH project, succeeds at providing these advanced analytical capabilities. We also demonstrate how to use this feature by providing documentation with examples. An extensive benchmark evaluation is also included to validate the performance of our novel *semantic multi-version concurrency control* mechanism and how it affects the data ingestion.

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 T5.3, at the final phase of the project (M30). The work that has been delivered during the duration of this task (M06-M30) was focused on the implementation of the brand new semantic multi-version concurrency control mechanism of the storage engine and its integration with the data management layer, in order to facilitate the provision of the *online aggregates,* which formulates the real-time analytics. We have achieved the integration with the INFINISTORE both at the storage layer and its relational query engine. In this deliverable, we report the rationale and motivation along with our approach and the design principles of our implementation. Additionally, documentation on how to use this mechanism in a declarative manner has been added, using both the storage’s direct API and the query engine. We validated the latter by integrating our solution with a running pilot of INFINITECH that relies its solution on the innovation that is brought as the outcome of this task. Finally, as the outcomes of this task have been already incorporated in LeanXcale datastore, we make use of some scenarios from one of the company’s established Proof-of-Concepts to evaluate the performance overhead using an extensive benchmark analysis.

1.2. Insights from other Tasks and Deliverables

The work that has been carried out in the scope of T5.3 (“Declarative Real-Time Data Analytics”) relies on the outcomes of the tasks of WP2, which define the overall user stories and requirements of the use cases of INFINITECH, and has been aligned with the INFINITECH RA that was previously introduced in D2.13 (“INFINITECH Reference Architecture – I”), under the scope of T2.7 (“Reference Architecture for BigData, AI and IoT in Financial Services Industry”). Apart from this, it relies on the outcomes of T3.1 (“Framework for Seamless Data Management and HTAP”), which implements the Hybrid Transactional and Analytical Processing (HTAP) framework that allows, on one hand, for data ingestion at very high rates, and on the other hand, the combination of operational with analytical workloads. The output of this task will be also beneficial to T3.3 (“Integrated Querying of Streaming Data and Data at Rest”) that implements the unified data query processing framework, which in turn allows the correlation streaming with batch processing. In particular, the provision of real-time analytics via the *online aggregates* can be used in query processing, as it allows for the execution of analytical operations in real-time, with the minimum latency possible. Finally, this task also gives input to T5.2 (“Incremental and Parallel Data Analytics”), as the latter depends on real-time analytics for the effective parallelization of its algorithms.

1.3. Updates from the previous version (D5.5)

In this version of this report we have added section 6, which provides an extensive benchmark analysis of our implementation when being stressed under heavy data ingestion workload. We made use of a real scenario coming from LeanXcale’s newly established Proof-of-Concepts with its potential clients, using the technology developed under the scope of this task. Then we evaluate the performance overhead of our implementation when using a vanilla data schema and one having the *online aggregates*, and we then discussed the benefits of the performance acceleration in the analytical query compared with the performance overhead we have to pay during the ingestion of the data.

1.4. Structure

This document is structured as follows: Section 1 introduces the document and section 2 provides the rationale and motivation of the Declarative Real-Time Analytical framework, including the problem stating and introducing the notion of *online aggregates* that this framework is based upon. Section 3 provides the documentation of this framework, while section 4 includes a hands-on demonstrator on how to use *online aggregates* in an application, providing code examples as guidelines for the application developers and data scientists. Section 5 demonstrates the use of the *online aggregates* in a declarative manner, both for their initial configuration that requires an extended SQL DDL syntax, and for the runtime execution that requires the submission of standard SQL statements. In this particular section, we validate our solution using pilot#2 ("Real-time risk assessment in Investment Banking") of the project and how we were able to improve the overall query execution time by several magnitudes of time. Section 6 provides the benchmark experimentation of our implementation and finally, section 7 concludes the document.

2 Motivation and Design Principles of Online Aggregates

This section describes the motivation and the design principles of the *online aggregates*. We describe a use case that is common in BigData applications in the finance sector, and we describe the fundamental pillar of our design: the *aggregate table*. After investigating why our approach is not feasible in traditional solutions relying either on relational SQL databases or NoSQL datastores, we explain the basic concepts of the *online aggregates* that make use of our semantic concurrency control mechanism and how our design solves the problems arising with other approaches and respect the *Snapshot Isolation* paradigm that our transaction processing mechanism follows.

2.1 A BigData application scenario as a reference example

BigData applications often require the support of data ingestion at high rates, where data can be collected from a variety of sources. Such sources can vary from data produced by IoT sensors installed in the soil or a vehicle and transmitted to an insurance organization, to the financial transactions of customers of a financial institution, records containing meta-information of a phone call collected by a telecom provider or data containing application metrics that is being acquired by a cloud provider in order to detect anomalies in the application behaviour for application performance monitoring.

For a financial institution, it might be crucial to monitor the maximum and average amount of money being used in a financial transaction in order to detect anomalies that might indicate fraud detection or money laundry operation from a client. Detecting those anomalies requires tracking the regular customer behaviour. Typically, this regular behaviour is modelled by several metrics: the maximum amount of money transferred in a single transaction, the average amount, the number of money transfers every given period of time and the overall amount being transferred. Additionally, there is a need to monitor and calculate these metrics at different aggregation levels: per customer, per region/area, per month etc.

In this example, we will describe how to compute this aggregation hierarchy in real-time, equivalently in terms of data consistency and in a cost effective way. Let's assume that there is a data table where each transaction of a client is being recorded. We will focus on two columns of that table, the name that identifies the client and the amount of money being transferred in a financial transaction. For instance, some sample rows of this table are illustrated in Table 1 below:

Table 1: Example of sample rows in a table containing transactions

Name	**Value**
Aleka	500
Pedro	250
José Maria	125
Aleka	200
Ricardo	480
Aleka	150
Ricardo	220

Large finance organizations like national banks or other similar institutions with millions of customers having different accounts have to keep track of all transactions that are being currently processed in their organizations. Depending on the size of the institutions, this might end up in the ingestion of hundreds of thousands of records per second. As the bank might have millions of customers to keep track of, and for each one of them it is required to keep track of the aforementioned different types of aggregations, the execution of each of those operations for each of its customers would be required, in a per-minute basis (or even less in case of money laundry and fraud detection cases). As each of those operations requires the traverse of the majority of the data table, this is very resource demanding from the database management system perspective.

2.2 Introducing Aggregate Tables

In order to reduce the cost of each particular aggregate operation and the overall cost of the analytical queries, one solution could be to compute those aggregations incrementally in a separate structure, whose values can be retrieved more efficiently. We call those structures *aggregate tables.* As a result, we could have an aggregate table per aggregation level. In our example, we could have one table containing the customer's summary amount of money transferred, the customer's maximum amount, etc. In real use cases, however, more aggregation levels might be needed for different periods and different aggregation levels. For instance, the financial institution might be interested to retrieve the average amount of money being transferred per customer and per month or day of the week, etc.

Sticking to our example, this *aggregate* table that contains the customer's summary amount of money transferred can be depicted in Table 2. This table has as key the name that identifies the customer. Then, every time a customer is performing a financial transaction, a new data item is being added in the structure depicted in Table 1, as part of the same database transaction, and the corresponding value in Table 2 is being updated according to its key. In the aggregate table, we would update the row with the associated name, updating the value by adding in the summary the new value. This will need to occur in each data insertion of the first table.

Table 2: Aggregate table containing a summary of transactions

Name	Value
Aleka	850
Pedro	250
José Maria	125
Ricardo	700

2.3 Problems Using Aggregate Tables with Traditional Approaches

As described above, our approach for solving the problem of cost-efficient execution of a real-time analytic operation, while data is being ingested at high rates, is based on the definition and use of the *aggregate table*. However, traditional datastore vendors still cannot benefit from such a data structure, for different reasons. This concerns both SQL databases and NoSQL data management systems. We will present why this is happening for both these two different datastore ecosystems.

We will first examine the NoSQL world. What happens if we implement the aggregate tables based on a NoSQL data store? Let's consider that we have two concurrent invocations over the same row in the aggregate table, for instance, the one that can be identified by the value '*Aleka*'. It is important to mention here that in real life scenarios deployed in production, it might be the case that millions of concurrent transactions might be inserting data over the same identifier, and as a result, trying to update the same row of the *aggregate table.* As NoSQL data store does not ensure data consistency in terms of database transaction semantics, in our example, both concurrent invocations will be allowed to proceed. However, this will create the effect that one of them will be lost in what is called the "Lost Updates" anomaly[1]. This is due to the fact that in order to update the *aggregate row* for the name 'Aleka', the process will first need to execute a *get* operation, using the 'Aleka' value as the key, retrieve the current value, and then execute a *put* operation with the same value as the key, and the aggregated value incremented with the corresponding value of the money that has been transferred.

To concretize our example, let's get back to the *aggregate table* of Table 2. Aleka has a summarized amount of money transferred with a value of 850. Now, two concurrent invocations appear for Aleka, with the amount of money transferred set to 150 and 200 respectively. As we are using a NoSQL datastore that does not ensure data consistency in terms of database transactions, both are allowed to be executed. As we explained, this will require each one of the two to perform a *get* operation, so that the current value 850 can be retrieved. The first invocation will have to calculate the accumulated value incremented by 150, and the second will increment this value by 200. This means that the first invocation will have to put the aggregate value of 1000 and the second one, the aggregate value of 1050. As both of them are being processed in parallel, let's assume that the second succeeds first, and thus updates the value in the *aggregate table* to 1050. Then eventually, the first invocation succeeds and updates the value to 1000. Now, the value of this aggregate row for 'Aleka' as the key, contains the value of 1000, instead of 1200, which is the accumulated summary of 850+150+200. This is caused due to what is called a *race condition* where two concurrent operations that share the same state (i.e. the value of the accumulated value 850 in the beginning) changes the value of the state in parallel and write it to the persistent storage. As a result, the second write operation erased the effect and the value of the first one. As *put* operations are blind writes, they do not take into account the value that was written before and just overwrite the latter.

This problem can be removed by using a traditional SQL database, which provides transaction semantics and ensures data consistency when concurrent invocations operate over the same data items. This happens using a multi-statement transaction, where an insert operation takes place on the table depicted in Table 1, and then an update operation modifies the value of the *aggregate table*. Concurrent transactions over the same data item (the aggregated row of Aleka) are managed by the transaction processing mechanisms of the database, and one has to be blocked until the successful commit of the other. The following pseudo-code can be used to describe such a transaction.

```
INSERT INTO Transactions (Name, Value) VALUES ('Aleka', 150)
DOUBLE currentvalue = SELECT FROM Aggregates WHERE Name = 'Aleka'
UPDATE Aggregates SET Value=currentvalue+150 WHERE Name='Aleka'
COMMIT
```

In this code snippet, the multi-statement transaction adds a new record in the first table with the value of 150 as the transfer, and then it reads the current value of the *aggregate table* for the corresponding table and updates its value accordingly, before commit. As the SQL databases use transactional concurrency control imposed by their transactional processing mechanism, the "Lost Updates" anomaly disappears. However, one has to take into account that there are two different families of implementations of the transactional concurrency control: locking used by the 2PL (two-phase locking) algorithms and multi-version, used by implementations relying on the *Snapshot Isolation* paradigm. Even if they are very different, both families are introducing an inherent contention problem. For instance, the transactional concurrency control forbids two concurrent transactions to modify the same row. This happens by allowing only one transaction to succeed and aborting (by a rollback) all other concurrent transactions that try to update the same data item. In our example, only the first database transaction that will try to eventually commit will succeed, and the second one will be aborted. This might probably be handled by the application by retry the insertion. In real use cases where hundreds or even thousands of concurrent transactions compete with each other to update a common data item, this will lead to a huge contention problem, no matter which implementation of the concurrency control algorithm the database vendor is making use of. In fact, relying on transactions to ensure data consistency and to remove the "Lost Updates" anomaly, will make our *aggregate table* proposition useless in use cases targeting Big Data needs.

2.4 Our solution: Online Aggregates

In order to solve this problem, we introduce at this phase the *online aggregates.* They are solving the problem with a new technology that is based on a novel semantic multi-version concurrency control provided by LeanXcale that has been currently integrated with the INFINITECH central data management layer. Relying on this new concurrency control, writes are not blind anymore. They actually carry the operation performed (i.e. sum(150) or sum(200) in our example). Since additions are commutative, they do not create write-write conflicts, as far as one keeps track that they are additions until the corresponding version of the row is written. Taking into account that this mechanism is built upon the operational mechanism of LeanXcale, integrated with the INFINITECH data management layer, this mechanism is making use of the *Snapshot Isolation* paradigm. As a result, in order to attain data consistency concerning this paradigm, we have implemented the multi-versioning algorithm in a sophisticated way. The underlying distributed storage layer is now able to support a new kind of data structure, using *aggregate columns* (we often call them *delta columns*), which adopt this new semantic multi-version concurrency control and enable *online aggregates*. This can be also combined with the HTAP capabilities implemented under the scope of T3.1 which allows for *online aggregates* under data ingestion at high rates, due to the capabilities of the HTAP framework to scale linearly in hundreds of nodes, which in practice, makes it possible to serve incoming workload independently of its rate.

In order to dive deeper into how the *online aggregates* work, one has to remember that the aggregate columns are conflict-less, and thus, they do not create write-write conflicts. As a result, we can have two different kinds of rows: regular rows (often called value rows), which are rows that contain values as in any traditional database management system, and operation rows (often called delta rows) that represent operations to be performed over the columns of the rows. In our previous example, these rows will have values sum(150), sum(200) etc. Regular rows create conflicts, as any regular row in a traditional database management system, however delta rows do not, as they are conflict-less. However, since they do not create conflicts the key problem that arises is that there might be gaps in the commit order, as concurrent transactions cannot commit in an ordered manner, and in fact, it will not be possible to generate the accumulated value for each delta at the time a transaction, with an operational row, is committed.

To further clarify this issue, let's go back to our previous example. As we rely on the *Snapshot Isolation* paradigm, each transaction is being given a timestamp. More information on how this works can be found in section 2.3 of the D3.1. In our example, we assume that the transaction that needs to add the value 150 is called t1, and the one that needs to add the value 200 is called t2. Let's assume that t2 commits first and t1 commits second. As opposed to the multi-version concurrency control mechanism, in order to provide different snapshots of the data set, each row is being labelled with a commit timestamp (CTS), which is a numeric value that is being increased monotonically and determines the order of the commit. In our example, as t1 commits second, it gets a CTS=2 and t2 gets a CTS=1. If a new transaction t3 is started after the commit of t1 and before the commit of t2, it would get a snapshot of 0, in order to guarantee that it will not observe any gap in the commit order. If it would get the snapshot at timestamp 2, it would miss the updates from the transaction with CTS 1, which would violate the *Snapshot Isolation*.

With our design, we solve the problem by converting operation (delta) rows into value rows only when the snapshot of the database is beyond the commit timestamp of the operation rows. This guarantees that the serialization order is gap free and therefore the generated versions are guaranteed to be consistent and according to the semantics of snapshot isolation. In the previous example, Table 2 contains a row (Aleka, 850, CTS=0). When the snapshot of the database reaches timestamp 1, then the operation row from t2 adding 200 will be converted into a value row generating the value row (Aleka, 200, CTS=1). When the snapshot reaches timestamp 2, then the operation row from t1 adding 150 is applied to the latest value row (the one generated by t2), yielding the following value row: (Aleka, 350, CTS=2). In this way, all the versions needed by the different snapshots are generated consistently, thus fulfilling the snapshot isolation semantics.

To conclude, we demonstrated how the proposed *online aggregates* can be used in order to return the online accumulated value of an aggregate operation, by enforcing data consistency and transactional semantics, removing the “Lost Updates” anomaly and avoiding the high contention that can be observed when using the same technique with traditional operational SQL database management systems. As a result and in combination with the HTAP framework developed under T3.1, they can be used to provide real-time analytics. At the first phase of the project, the initial design had been validated and we had provided a first implementation of our solution. The next section of this document reports on the use of this functionality, focusing on how this can be invoked in a declarative fashion. In the current version of the deliverable, we have also included a separate section with more technical details of the implementation of the *online aggregates*, which provide the declarative real-time analytical framework of INFINITECH. This now relies on the relational query engine of the INFINISTORE that has been already integrated with our proposed solution at this phase of the project. It is important to highlight that the implementation details of our novel semantic concurrency control mechanism described in this section has been filed by LeanXcale for a patent, and cannot be reported in a public document.

3 Using Declarative SQL to enable Real-Time Data Analytics

This section provides basic documentation on how to configure the use of the *online aggregates* and how an aggregate operator can be invoked by the application of a data analyst. In order to configure a table to exploit the *online aggregations,* the database administrator would need to declare this online aggregation table that is related to a raw table. We will continuously refer to the table containing the raw data with the regular rows as *parent table.* The following code snippet provides an example on how to declare those tables.

```
CREATE TABLE EVENTRAW (
  ev_id integer NOT NULL,
  ev_im_id integer,
  city char(24),
  ev_price integer,
  ev_data char(50),
  CONSTRAINT pk_eventraw PRIMARY KEY (ev_id)
);

CREATE ONLINE AGGREGATE ON EVENTRAW AS AGG_ EVENTBYCITY (
  city,
  max_price max(ev_price),
  count_price count(*),
  min_price min(ev_price),
  sum_price sum(ev_price)
);
```

In this example, we define a *parent* table called EVENTRAW, with a standard DDL statement. This declares that this table will have the *ev_id* as its primary key, and further defines four (4) additional columns containing raw data. Additionally, it defines the AGG_EVENTBYNAME as an *aggregated table* on the parent one defined above, with the definition of the row operators. We can notice that the *city* will be the key of this table, as it does not contain any function, while the 4 additional columns will calculate online the values of the max_price, count_price, min_price and sum_price. From the code snippet, it can be noticed that these columns will calculate online the result of the aggregate operation (max, min, count, sum) of the column ev_price, which is included in the *parent* table.

After defining the *aggregate table,* the user might want to start inserting some data in those tables. A data insertion in the *parent table* will need to be accompanied by a corresponding insertion in the *aggregate table.* In order for these two statements to be atomic and to ensure data consistency when updating both tables, these statements need to be bracketed inside a single transaction. The following code snippet provides such an example.

```
UPSERT INTO EVENTRAW VALUES (1, 11, 'London', 10, 'aabbccdd');
UPSERT INTO AGG_EVENTBYCITY VALUES ('London', 10, 1, 10, 10);
COMMIT;
```

In this transaction, a new record is being added in the EVENTRAW table, with the corresponding values. At the same transaction, the record with key 'London' is being updated, by an UPSERT in the corresponding *operational row.* In this example, three (3) columns are calculating the value based on an aggregate operation on the ev_price column, and as a result, the UPSERT statement adds the value 10. This will be translated in operational values as max(10), min(10), sum(10) and the actual result of this operation will be calculated online. As the count_price delta column is operating on a count(*), for this column we added the '1'. It is important to mention at this point that the insertion on a *parent table* does not need to be followed programmatically anymore with an UPSERT on all their relevant *aggregated tables* with which it is related. The advancements that took place during the second phase of the project removed this necessity and the query engine itself can now identify that there is an insertion over a parent table that is associated with a list of *online aggregates* and it will add the corresponding values transparently to the data user. As a result, it is not up anymore to the application developer or data scientist to write these lines of code appropriately, that they can be error-prone. In our current version of the prototype of the *Declarative Real-Time Analytics Framework*, this has been made automatically by the framework itself, as it has been planned to be implemented during this second phase of the project. However, we keep the syntax here as it can be more intuitive for the reader to understand these concepts. We have included a specific demonstrator in a separate section to highlight how everything works together now.

No matter how the *operational* row is being added to the *delta table,* the important innovation of our approach is that we rely on the semantic concurrency control to ensure data consistency while executing these two statements, while at the same time removing the high contention that they can introduce when being executed within a traditional SQL relational database management system. Let’s have a look at the following code snippet, which illustrates how the multi-statement transaction defined above should be re-written using a typical SQL database.

```
UPSERT INTO EVENTRAW VALUES (1, 11, 'London', 10, 'aabbccdd');
UPDATE AGG_EVENTBYCITY SET
  MAX_PRICE=MAX(MAX_PRICE, 10),
  COUNT_PRICE=COUNT_PRICE+1,
  MIN_PRICE=MIN(MIN_PRICE, 10),
  SUM_PRICE=SUM_PRICE+10
WHERE CITY = 'London';
COMMIT;
```

In the case above, according to the type of database, it can cause several problems. As explained in the previous section, NoSQL databases will suffer from the “Lost Updates” anomaly that will be revealed when executing the update statement in the AGG_ EVENTBYCITY. In particular, the relevant {attribute} = {attribute} + value expressions will require a *get* operation, followed by a *put*, which will cause the race condition when two or more concurrent invocations occur, which is actually the cause of the “Lost Updates”. Relational SQL databases, on the other hand, will suffer from huge contention, as concurrent updates will target the same operational row identified by the key ‘London’. On the contrary, by managing these types of operations using semantic concurrency control, there is no contention. The only impact will be on data ingestion, as it will require internally two operations instead of one, however, the overhead is very low and the overall impact on getting the pre-calculated values of the aggregated operations is significantly much more important.

Let’s investigate now how the application developer or the data analyst will have to perform an aggregate operation on the *parent table* that contains the original raw data. In order to get the max and average price per city, he/she should submit the following SQL statement:

```
SELECT city, MAX(ev_price), AVG(ev_price) FROM EVENTRAW GROUP BY city
```

This statement will indeed return the maximum and minimum prices, grouped by the city. However, this is a cost-demanding operation, as the database management system will have to traverse the whole data table, in order to retrieve all *ev_prices* for the cities, group them by the city column, and then apply the operation. As this will require the traverse of the whole table, its latency is significant high, and as a fact, it cannot be considered as online, neither can it be used in use cases where the online calculation of these values is crucial. Additionally, in traditional operational datastores based on the 2PL (two-phase locking) protocol for ensuring transactional semantics, it would require the database management system to acquire *shared locks* in each data item that is being accessed. This would prevent any other write operation to be performed in parallel, as all data modification operations need to acquire an *exclusive lock*, which will be forbidden, as *shared locks* have already been put by the aggregate operation.

Using the *online aggregates* proposed by INFINITECH's *Declarative Real-Time Analytics* framework, that statement will be internally translated to the following:

```
SELECT city, MAX_PRICE, SUM_PRICE/COUNT_PRICE FROM AGG_EVENTBYCITY
```

This second query runs in much lesser time compared to the previous one, whose execution time is highly increased due to the need of traversing a lot more rows. Even if we apply a filter condition, in order to get the maximum and average price of a specific city (i.e. adding a WHERE city='London') in the first example, it will require the traverse/scan of the data table, which has a complexity of O(n) or O(log(n)) (according to whether or not the column *city* is indexed). On the contrary, using online aggregates, the *online aggregate* operation costs O(1) according to the theory of complexity. We need to highlight at this point that using the advancements we have accomplished during the second phase of the project, we do not need to *attack* the *delta table* anymore, but use an SQL statement directly *attacking* the parent one instead. The query engine will identify that the aggregate operation is linked with an *online aggregate* and transform the query plan accordingly. More information on this approach will be given in a separate section.

Our approach can be compared to calculating those values in memory, so that you can have the results already pre-calculated and have the results returned immediately. However, as we are dealing with persistence, doing those operations in memory cannot be done. There are several issues regarding data persistency and fault-tolerance in cases of unexpected shutdowns or crashes, concurrency control when having to deal with high rates of parallel ingestion and most importantly, memory is not infinite, and therefore, there is a limited size of records that such a memory-based implementation could handle. Instead of using an in-memory implementation, we push all those issues to be solved by the data management layer, which has been designed to solve those and provides high availability, fault-tolerance and concurrency control mechanisms, while it uses the persistent storage volumes to scale out.

To conclude, the *online aggregates* are a really powerful mechanism because they allow you to have pre-computed data immediately available to serve your application. Note that even if the data analyst needs a complex KPI, this may be composed of pre-computed aggregates. That's the case of the standard deviation statistic for example. Other common scenarios relevant to the finance and insurance institutions can be also foreseen. For instance, *online aggregates* can be used in cases where there is the need for an immediate statistical aggregate in order to provide real-time results for an application. This is the case of the risk assessment use cases of INFINITECH. Moreover, working with multi-resolution data is also a good example. There might be a system whose source devices send information every second, but most statistics can be calculated much more easily at minute resolution. Aggregates can be used to store aggregated data at minute resolution and at 15 minute resolution will have second raw data. This is the case of the insurance pilot#11 of INFINITECH, which relies on IoT data coming from sensors installed in the vehicles, and they are pre-processed before being sent to the central application component deployed within the sandbox. Finally, together with multidimensional partitioning, online aggregates can be seen as pre-computed Online Analytical Processing (OLAP) cubes. This is a very powerful data to have available with little latency to get it. The following section provides a hands-on example on how to make use of this framework.

4 Real-Time Data Analytics in practice using the direct API

In this section we will provide a hands-on example on how to make use of the *Real-Time Data Analytics* framework, developed in the context of T5.3, based on a hypothetical scenario of an application that monitors the web traffic generated by end-users visiting a website. The requirement is for the business analyst to have a dashboard that he/she can be able to analyse the user visiting the website every second. In case the system architect does not want to implement a typical but complex lambda architecture [4], but instead, to rely only on a single database management system, then it would be required to periodically execute count operations over a group-by clause at the time of the insertion. However, this would cause the loss of performance due to the fact that *reads* are competitive with *writes* in an operational datastore. As there are usually many more *insertions* than *reads*, a solution could be to cache the results of the analytics in memory, yet with all the drawbacks explained in the previous chapter. Instead, the system architect could decide to exploit the *online aggregates* provided by our framework in order to serve those aggregate operators in real-time. In this example we will make use of the direct API of INFINISTORE that exposes the functionalities related to the *online aggregates.* This was the standard way for using this feature at the first phase of the project. We keep this chapter in the current version of the report as it provides an intuitive way for the user to understand the deeper details of our approach. In the next chapter, we also provide an additional example on how to make use of the *online aggregates* using standard SQL statements and taking advantage of the advancements that we did during the second phase in the layer of the query engine.

Online aggregates are built on top of the transactional and analytical processing (HTAP) provided by the data management layer in the scope of T3.1, which is the fundamental pillar for those operations. Our approach makes use of the *delta column*, which is capable of providing the result of an aggregation query, such as the one described above, pre-calculated at the time of insertion, and persistently stored in the storage medium. This way, getting the aggregate for a value requires simply reading a row from the relevant aggregate table, which is already pre-calculated, instead of doing a scan to find the right row and calculate the aggregation. This, in turn, means aggregations in real-time. They enable the calculation of aggregates of any kind over the data management layer, without executing a heavy group-by query and without losing performance on an insertion. This example will demonstrate this, which simulates a monitoring application.

As explained above, when implementing these types of applications, it is very important to take into consideration the balance between two things: the performance of the data insertions and the query execution. In the majority of the cases, the number of insertions is dominant, as the rate can be hundreds of thousands of records per second, while a read operation is executed periodically. As a result, it seems reasonable to prioritize the insertion performance. However, as our hypothetical application would need to report various things that would need to execute aggregations over the end-user's cookie IDs, when having high volumes of data, those aggregations are very low, and they can block the insertions. This means that the system is either going to lose currently inserted data or that the reported data are not going to be real.

4.1 Setting up the application

The application that will demonstrate the use of the *online aggregates* consists of the following components:

- The INFINITECH data management layer.
- The Declarative Real-Time Analytics framework, which makes use of the data layer.
- A program[3] that simulates data insertion and executes queries periodically.
- An SQL client, to visualize the results of the queries (SQuirreL [2]or DBeaver [3], are recommended).

Both the INFINITECH data management layer and the Declarative Real-Time Analytics framework are available in the project's private repository and will be retrievable by the INFINITECH Marketplace.

For our hands-on demonstrator, we will generate some controlled data to simulate the insertion of thousands of cookies per second. An example table containing information of the user when visiting a website can be depicted as follows:

Table 3: Table containing information from a visitor's cookies

Cookie ID	Post Date	Other fields
c034868a-2e3e-11eb-adc1-0242ac120002	2020-11-17 15:48:52	{value_1}
c0348a36-2e3e-11eb-adc1-0242ac120002	2020-11-17 15:48:53	{value_2}
c0348b44-2e3e-11eb-adc1-0242ac120002	2020-11-17 15:48:53	{value_3}
c0348c16-2e3e-11eb-adc1-0242ac120002	2020-11-17 15:48:54	{value_4}

The architecture of the solution of this hypothetical scenario consists of i) a loader process and the program that performs the query executions, ii) the INFINITECH data management layer, with the addition of the Advanced Analytical Capabilities provided by the Declarative Real-Time Analytics framework and iii) an SQL client, which can be one of the proposed ones above. We can benefit from the versatility of the data management layer itself, which provides various ways for data connectivity, using an SQL or a No-SQL interface. In our example, we rely on the No-SQL interface for both insertion and query execution, in order to benefit from its improved performance, as it bypasses the query engine of the datastore, however, there is always the possibility of executing SQL queries, and we can make use of the SQL client for that. Having this versatility to make use of the *online aggregates*, makes it possible to use them with whatever Business Intelligence (BI) tool the data analyst is familiar with or is better for his / her purposes.

Regarding the data model, we will create a table similar to the one depicted in Table 3 in order to store the cookie information and two additional tables to store the delta aggregates that will be pre-calculated at the time of the insertion.

[3] https://gitlab.com/leanxcale_public/onlineaggregation

4.2 Implementing the application using online aggregates

Firstly, we will implement the data inserter program, which will be built using Java and Spring Boot. It will respond to the following three types of requests:

- Run: to start the insertion.
- Clean: to clear the database.
- Query: to start the periodic query executor.

It will make use of a CSV file where it will start reading its lines and it will ingest them into the database.

Before starting, the application developer needs to make use of the No-SQL interface to enable the data connectivity with the data management layer. Using maven, it can be locally installed in the maven repository and then it can be used as a dependency. The following code snippet depicts this process:

```
mvn install:install-file -Dfile=kivi-api-1.6-direct-client.jar -
DgroupId=com.leanxcale -DartifactId=kivi-api -Dversion=1.6 -Dpackaging=jar

<dependency>
   <groupId>com.leanxcale</groupId>
   <artifactId>kivi-api</artifactId>
   <version>1.6</version>
</dependency>
```

Now it is the time to create the main table containing the row data from the end-users visits, and the corresponding *aggregate tables.* The main table will have an alphanumeric ID, and a timestamp, which will form a composite primary key. In standard SQL this can be done by executing the following statement:

```
create table INFO (
  ID VARCHAR,
  POSTDATE TIMESTAMP,
  OTHERFIELD  VARCHAR,
  CONSTRAINT PK_INFO PRIMARY KEY (ID, POSTDATE)
);
```

Programmatically via the No-SQL interface (or as we call it *direct API*), this can be achieved by executing the following code snippet:

```
private static final String TABLE_NAME = “INFO”

(…)

Settings settings = new Settings();
settings.credentials(new Credentials()
.setDatabase(databaseName)
.setUser(user)
.setPass(password.toCharArray()));
settings.transactional();

try (Session session = SessionFactory.newSession(URL, settings)) { // New session
// Table creation:
if (session.database().tableExists(TABLE_NAME)) {
       // Primary key fields
        List<Field> keyFields = Arrays
                    .asList(new Field[]{new Field(“id”, Type.STRING),
                            new Field(“postdate”, Type.TIMESTAMP)});
```

```
             // Rest of fields
             List<Field> fields = Arrays
                     .asList(new Field[]{
                                 new Field("otherfield", Type.STRING)
                                 });
             // Table creation
             session.database().createTable(TABLE_NAME, keyFields, fields);
       }
}
```

In order to retrieve the information regarding how many different users have visited the monitored web page and how many times, or how many visits per day our page has had, then the database administrator will have to execute SQL queries such as the following:

```
select id, count(id) from info  group by id;

select FLOOR(POSTDATE to DAY) as fecha, count(*) AS total FROM info group by
FLOOR(POSTDATE to DAY);
```

To take advantage of the Declarative Real-Time Analytics framework and execute these queries online in order to get the pre-calculated aggregated value with the same cost of accessing a row by its primary key, we need to use the *delta* columns to holds operational values. Those operational values will pre-calculate the result of the aggregate operation. For that, we are going to create an *aggregate table*. We would need to define two: one for the ID count and one for the date count. The field from which we are going to aggregate must be the aggregate table PK, and the aggregator must be defined as delta in the field creation. The following code snippet indicates how to do it:

```
private static final String ID_COUNT_DELTA_TABLE_NAME = "INFO_ID_DELTA";
private static final String DATE_COUNT_DELTA_TABLE_NAME = "INFO_DATE_DELTA";

(…)

Settings settings = new Settings();
settings.credentials(new Credentials()
.setDatabase(databaseName)
.setUser(user)
.setPass(password.toCharArray()));
settings.transactional();

try (Session session = SessionFactory.newSession(url, settings)) { // New session

// Aggregation table creation:
if (!session.database().tableExists(ID_COUNT_DELTA_TABLE_NAME)) {
   List<Field> keyFields = Arrays
           .asList(new Field[]{new Field("id", Type.STRING)});
   List<Field> fields = Arrays
           .asList(new Field[]{
                   new Field("count", Type.LONG, DeltaType.ADD)});
   session.database().createTable(
           ID_COUNT_DELTA_TABLE_NAME, keyFields, fields);
}
// Aggregation table creation:
if (!session.database().tableExists(DATE_COUNT_DELTA_TABLE_NAME)) {
   List<Field> keyFields = Arrays
           .asList(new Field[]{new Field("postdate", Type.DATE)});
   List<Field> fields = Arrays
           .asList(new Field[]{
                   new Field("count", Type.LONG, DeltaType.ADD)});
   session.database().createTable(
          DATE_COUNT_DELTA_TABLE_NAME, keyFields, fields);
}
```

We have defined two tables, one per aggregate:

- The first table, INFO_ID_delta, corresponds to query select id, count(id) from info group by id;. Since we want to group it by ID, the ID is going to be the PK of this delta table. The aggregate will be the count, and we will add 1 to the pre-calculated aggregate (please note the delta field is declared as DeltaType.ADD).
- Similarly, the second table, INFO_DATE_delta, corresponds to query select FLOOR(POSTDATE to DAY) as fecha, count(FLOOR(POSTDATE to DAY)) AS total FROM info group by FLOOR(POSTDATE to DAY); the table PK is going to be the postdate field, and the aggregate will again be the count.

After the schema and the *aggregated tables* definition, we are going to write a code to insert data from the CSV file. It is important to highlight at this point that an insertion in the main table INFO must be followed by a corresponding insertion in the *aggregated tables* that we previously defined. The following code snippet illustrates this:

```
try (Session session = SessionFactory.newSession(url, settings)) { // New session
     session.beginTransaction();
     Table infoTable = session.database().getTable(TABLE_NAME);
     Table infoIdTable = session.database().getTable(ID_COUNT_DELTA_TABLE_NAME);
     Table infoPostdateTable =
           session.database().getTable(DATE_COUNT_DELTA_TABLE_NAME);

     // Insert into info table
     Tuple tuple = infoTable.createTuple();
     tuple.putString("id", id);
     SimpleDateFormat dateFormat = new SimpleDateFormat(
                                          "dd-MM-yyyy HH:mm:ss.SSS");
     Date date = dateFormat.parse(postdateString);
     tuple.putTimestamp("postdate", new Timestamp(date.getTime()));
     tuple.putString("otherfield", otherfield);
     infoTable.insert(tuple);

     // Insert into info_id_delta
     Tuple tupleIdDelta = infoIdTable.createTuple();
     tupleIdDelta.putString("id", id);
     tupleIdDelta.putLong("count", 1L); // We add 1 to our DeltaType.ADD field
     infoIdTable.upsert(tuple);

     // Insert into info_date_delta
     Tuple tupleDateDelta = infoPostdateTable.createTuple();
     tupleDateDelta.putDate("postdate", new java.sql.Date(date.getTime()));
     tupleDateDelta.putLong("count", 1L);
     infoPostdateTable.upsert(tupleDateDelta); // We add 1 to our DeltaType.ADD
                                                 field

     session.commit();
}
```

Finally, in order to retrieve data using the *online aggregates,* the following code snippet can be used, which relies on the No-SQL interface that we provide.

```
Try{
   session= SessionFactory.newSession(url, settings);
   infoTable = session.database().getTable(Constants.TABLE_NAME);
   infoIdTable = session.database().getTable(Constants.ID_COUNT_DELTA_TABLE_NAME);
   infoPostdateTable = session.database().getTable(Constants.DATE_COUNT_DELTA_TABLE_NAME);
   while (executions < 1000) {
       log.info("‑-------------------------------------------------------------------------------------
------------------------");
       log.info("LX: Querying info table for number of rows”…");
       session.beginTransaction();
       // Build TupleIterable, execute find with count aggregation and iterate
       // over the result (select count(*) from info)
       TupleIterable res = infoTable.find().aggregate(Collections.emptyList(),
Aggregations.cou‟t("numR”ws"));
       res.forEach(tuple -> log.in‟o("Row‟: " + tuple.getLo‟g("numR”ws")));
       log.in‟o("LX Querying info table do”e!");


       log.in‟o("-------------------------------------------------------------------------------------
------------------------");
       log.in‟o("LX: Querying id delta table”..");
       long t1 = System.currentTimeMillis();
       // select * from info_id_delta
       TupleIterable res2 = infoIdTable.find();
       res2.forEach(tuple -> log.in‟o("I‟: " + tuple.getStri‟g(”id")‟+ " Coun‟: " +
tuple.getLo‟g("co”nt")));
       long t2 = System.currentTimeMillis();
       log.in‟o("LX Query time: {}”ms", ‑2 - t1);
       log.in‟o("LX Querying id delta table do”e!");

       log.in‟o("-------------------------------------------------------------------------------------
------------------------");
       log.in‟o("LX: Querying postdate delta table”..");
       long t3 = System.currentTimeMillis();
       // select * from info_date_delta
       TupleIterable res3 = infoPostdateTable.find();
       res3.forEach(tuple -> log.in‟o("Dat‟: " + tuple.getDa‟e("postd”te")‟+ " Coun‟: " +
tuple.getLo‟g("co”nt")));
       session.commit();
       long t4 = System.currentTimeMillis();
       log.in‟o("LX Query time: {}”ms", ‑4 - t3);
       log.in‟o("LX Querying id delta table do”e!");

       executions++;
       Thread.sleep(5000);
   }
}
finally {
   if (session == null) {
       session.close();
   }
}
```

To conclude, with this hands-on chapter we demonstrated how we can implement a hypothetical application that could benefit from the use of *online aggregates* and, also, the capabilities of the Declarative Real-Time Analytical framework of INFINITECH. During this second phase of the project, we have accomplished to extend the SQL syntax which now provides automation in the query engine level, so that the definition of *table aggregations* and insertions in tables that trigger insertions on *aggregated tables* can be automatically executed in the query engine level. This will be demonstrated in the following chapter.

5 Real-Time Data Analytics in practice using SQL syntax

During the first phase of the project, the main focus of the work that was carried out under the scope of task T5.3 “Declarative Real-Time Data Analytics” was to implement the fundamental pillars for the provision of the *online aggregates.* This required the implementation of our novel semantic concurrency control mechanism in the storage engine of INFINISTORE with the definition of the *operational rows* that are stored in the *aggregated/delta* tables, using *delta columns* over numeric data types. Moreover, we exposed the relevant functionalities by extending the *direct API* of the INFINISTORE, which allows the data user to define *delta tables*, add data to these new data structures and perform operations that can benefit from the *online aggregates.*

The main drawback of this approach is that it is the responsibility of the data user or application developer to take care of the insertion of both the data value row and the *operational* row to the two data structures (the parent and the derived/delta table), which can be error prune, in case he or she forgets to add both. Moreover, in order to make use of the *online aggregates,* the data user or application developer needed to know the underlying specifics of the data structures in order to access them and perform those operations. Finally, the use of the *direct API*, which is not standard, introduced the need to write a data vendor-specific code at the application level, while it could not be exploited by analytical frameworks that often require data connections provided via well-known standards such as JDBC, while statements can be auto-generated using standard SQL statements.

For all these reasons, during this second phase of the project, the main focus was given on the integration of the relational query engine of INFINISTORE with the storage engine, in order to provide real-time data analytics using our *online aggregates* in a declarative manner using standard SQL, while the query engine will take the responsibility of executing the involved operations transparently to the data user, thus hiding all internal specifics of the framework from him or her. The latter will only have to declare the *online aggregates* data tables, and then only focus on writing the queries to insert or retrieve data with standard SQL statements.

This section provides examples on how to configure and make use of this framework for declarative and real-time analytics, diving into detail on how everything is being executed in the background by the relational query engine of INFINISTORE. This was successfully demonstrated during the first interim review of the project, integrated by Pilot#2 “Real-time risk assessment in Investment Banking”, which is the pilot that benefits from our solution.

5.1 Online Aggregates in Real-Time Risk Assessment

Pilot#2 “Real-time risk assessment in Investment Banking” calculates the risk for making a trade in order to move an amount of money from one financial currency to another at a specific time. For the analysis of the risk assessment, it gathers data from different data streams that contain information of the current value of a *product* (e.g., EURO to US Dollars currency) every second. This data is being ingested into the INFINISTORE. As there is a vast amount of different products, this can end to hundreds of data items needed to be ingested per second, which is a fairly intensive one. Each data row contains the type of product, the *open, high, low, close, up* and *down* values of the product at a specific timestamp. An example of these rows can be depicted in the following code snippet:

```
1.05389,1.05389,1.05389,1.05389,0,0,2020-03-19 23:05:24,EURCHF
1.06887,1.06887,1.06887,1.06887,0,0,2020-03-19 23:05:24,EURUSD
1.06885,1.06885,1.06885,1.06885,0,0,2020-03-19 23:05:25,EURUSD
1.14986,1.14986,1.14986,1.14986,0,0,2020-03-19 23:05:25,GBPUSD
1.05389,1.05389,1.05389,1.05389,0,0,2020-03-19 23:05:27,EURCHF
```

```
                    1.06886,1.06886,1.06886,1.06886,0,0,2020-03-19 23:05:27,EURUSD
```

In this scenario, pilot#2 needs to feed its AI algorithm with aggregated values of this data that contains the average values per product and per a specific time period (i.e. minutes, hours, days). In order to do so, it would need to push these statements down to the data management layer, so that the latter can do the pre-processing and send only the aggregated values to the AI algorithm. This would require a *scan* operation on a vast amount of data items stored in the table that contains this information, and as we have seen, this not only is cost-expensive but also contradictive when performing this type of operation on a live dataset which is continuously being loaded with new information. This is exactly what the *online aggregates* solve and we will see in the following subsections how we can make use of them in this scenario.

5.2 Deploying Online Aggregates with INFINISTORE

At this phase of the project, the implementation of the *online aggregates* has been integrated with INFINISTORE, and can be used by accessing the latter via JDBC connections. As a result, we would need to deploy the INFINISTORE data management layer, as the *online aggregates* are an integral part of this solution. Right now, the INFINISTORE can be deployed via the INFINITECH way defined under the scope of WP6, making use of its relevant blueprints. The deployment will make use of Kubernetes, and the following code snippet depicts a blueprint YAML file that can be used[4]:

```
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:
          - name: infinistore-datasets-storage
            mountPath: /datasets
      containers:
        - image: harbor.infinitech-h2020.eu/data-management/infinistore:latest
          name: infinistore
          ports:
            - containerPort: 2181
```

[4] The blueprints for pilot2 can be also found under the project's gitlab repository: https://gitlab.infinitech-h2020.eu/blueprint/pilot2

```
            - 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 1529 port, which is where JDBC is listening to:

```
apiVersion: v1
kind: Service
metadata:
  name: infinistore-service
  labels:
    app: infinistore
spec:
  ports:
    - name: "1529"
      port: 1529
      targetPort: 1529
  selector:
    app: infinistore
```

Using these blueprints, the INFINISTORE can be deployed using Kubernetes and can be accessible from an SQL client like DBeaver. In the following section we will see how to declare the *online aggregates* for the user scenario we are exploring for pilot#2.

5.3 Declaring the Online Aggregates

As it was shown, the raw data that contains the financial information per product consists of 6 numeric values, an identifier and a timestamp and it will be imported to a single data table. The identifier with the timestamp will formulate the compound primary key of that table. The following DDL statement creates this structure:

```
CREATE TABLE TickData (
TIK_OPEN DOUBLE,
TIK_HIGH DOUBLE,
TIK_LOW DOUBLE,
TIK_CLOSE DOUBLE,
TIK_UP DOUBLE,
TIK_DOWN DOUBLE,
DATETIME TIMESTAMP,
PRODUCT VARCHAR,
PRIMARY KEY(PRODUCT, DATETIME)
);
```

This table uses a standard DDL SQL statement that declares a data table named *TickData* with 6 numeric columns of type *DOUBLE* and two additional ones that are part of the primary key. This will be the *parent* table of the *online aggregates* that we will declare soon after. The AI algorithm of pilot#2 needs to retrieve the average *close* value per product per day, hour and minute. Instead of executing these three statements by directly *attacking* the parent table, we can now make use of the *online aggregates.* That way, we will avoid traversing a vast amount of data, and we can now rely on our novel mechanism that pre-calculates those values. As we need three types of aggregations, we will create three *online aggregates*.

The following code snippet declares an *online aggregate* per product and per day:

```
CREATE ONLINE AGGREGATE MIN_CLOSE_DAILY AS
sum(TIK_CLOSE) MIN_CLOSE_SUM,
count(TIK_CLOSE) MIN_CLOSE_COUNT
FROM TICKDATA
GROUP BY PRODUCT,
CTUMBLE(DATETIME, INTERVAL '1' day, TIMESTAMP '1970-01-01 00:00:00') DATETIME;
```

The following code snippet declares an *online aggregate* per product and per hour:

```
CREATE ONLINE AGGREGATE MIN_CLOSE_DAILY AS
sum(TIK_CLOSE) MIN_CLOSE_SUM,
count(TIK_CLOSE) MIN_CLOSE_COUNT
FROM TICKDATA
GROUP BY PRODUCT,
CTUMBLE(DATETIME, INTERVAL '1' hour, TIMESTAMP '1970-01-01 00:00:00') DATETIME;
```

The following code snippet declares an *online aggregate* per product and per minute:

```
CREATE ONLINE AGGREGATE MIN_CLOSE_DAILY AS
sum(TIK_CLOSE) MIN_CLOSE_SUM,
count(TIK_CLOSE) MIN_CLOSE_COUNT
FROM TICKDATA
GROUP BY PRODUCT,
CTUMBLE(DATETIME, INTERVAL '1' minute, TIMESTAMP '1970-01-01 00:00:00') DATETIME;
```

As we can see, we have extended the DDL SQL syntax of the INFINISTORE relational query engine to allow the definition of such structures. Now the reserved words *ONLINE AGGREGATE*, followed by its name will instruct the query engine to create a derived table that will be associated with the *parent table* that is defined in the FROM clause. In our scenario, that will be the *TickData* table, previously declared.

Our structures will pre-calculate the results of the count and sum operations over the data column *TIK_CLOSE* of the parent table. We need to remember that the data analyst is interested in finding the average value of this column per product and per different time period. The average is equivalent to the summary divided by count, so we define these operations in our *online aggregates*.

Last but not least, the columns that are involved in the GROUP BY clause will formulate the primary key of our new data structure. In our scenarios we rely on the *product*, which is a static data column defined in the parent table, and the result of an SQL function, whose result is being retrieved dynamically by applying the CTUMBLE function over another static column of the parent table, that is also part of the primary key: the DATETIME. This is a feature that was implemented at the very end of the second phase of this task and allows to define an *online aggregate* not only over static columns but also over dynamic ones, as this example clearly demonstrates.

In this subsection, we saw how we can make use of the declarative real-time analytical framework to define our *online aggregates* using extended SQL DDL statements over standard JDBC connections. As we have created our data structures, we load the parent tables with a set of historical data containing financial currencies for a list of different products. The data has been provided to us in a CSV file and we made use of the INFINISTORE data loader to import them, whose use is out of the scope of this report. The important thing to highlight here is that although we use the data loader that only adds raw data to the parent table, we do not need to proceed to any further action to update the *delta table*, as this is being done automatically by the storage engine layer and transparently to the data user. Having loaded the historical dataset, in the next subsection we will demonstrate the use of the *online aggregates* in practice.

5.4 Using Online Aggregates with standard SQL statements

To highlight the benefits of using our framework for declarative real-time data analytics, we deployed two instances of the INFINISTORE: in the first one we used only the parent table, while in the second we also declared the three online aggregates. Then we load both tables with the same CSV file that contains the historical data for financial currencies. The difference is that in our second deployment, this data was transparently loaded in the newly created derived tables.

We used the DBeaver tool to connect to both instances. We will call the first instance *Vanilla Infinistore* and the second *Online Aggregates.* When connecting to both, we can see that the vanilla one contains only the TickData table, along with some additional ones that are used by pilot#2, while the second instance also contains the newly created *online aggregates.* This is depicted in Figure 1.

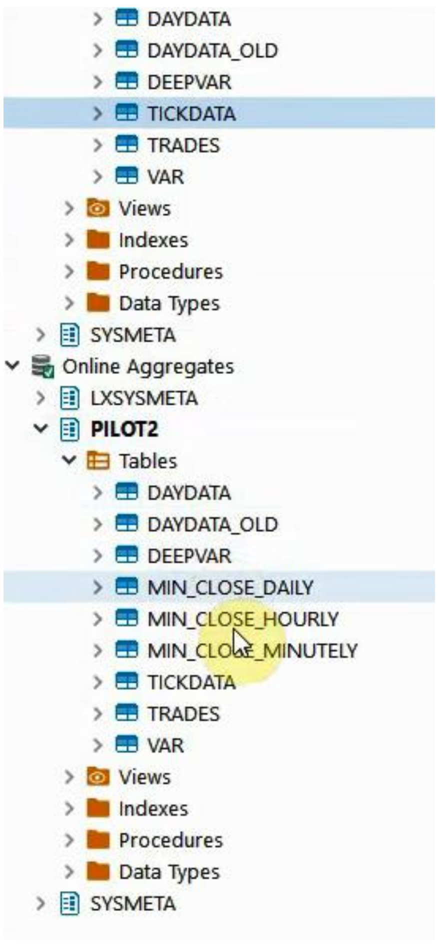
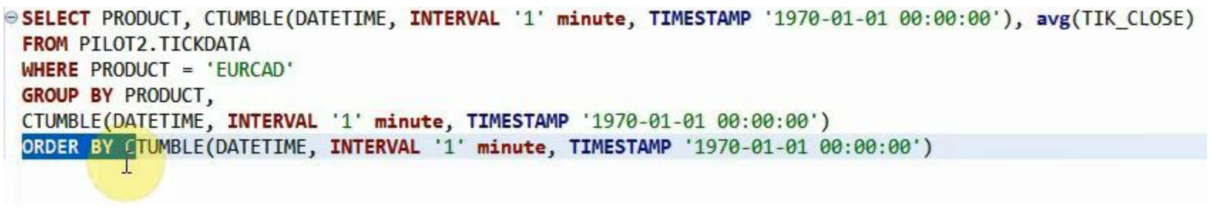


Figure 1: Online Aggregates Data Structures

Now we will execute a query that is being periodically sent by the AI algorithms of pilot#2, to get the average value of the *TIK_CLOSE* per product and per minute. The query can be depicted in Figure 2.



```
SELECT PRODUCT, CTUMBLE(DATETIME, INTERVAL '1' minute, TIMESTAMP '1970-01-01 00:00:00'), avg(TIK_CLOSE)
FROM PILOT2.TICKDATA
WHERE PRODUCT = 'EURCAD'
GROUP BY PRODUCT,
CTUMBLE(DATETIME, INTERVAL '1' minute, TIMESTAMP '1970-01-01 00:00:00')
ORDER BY CTUMBLE(DATETIME, INTERVAL '1' minute, TIMESTAMP '1970-01-01 00:00:00')
```

Figure 2: SQL Aggregation Query Statement

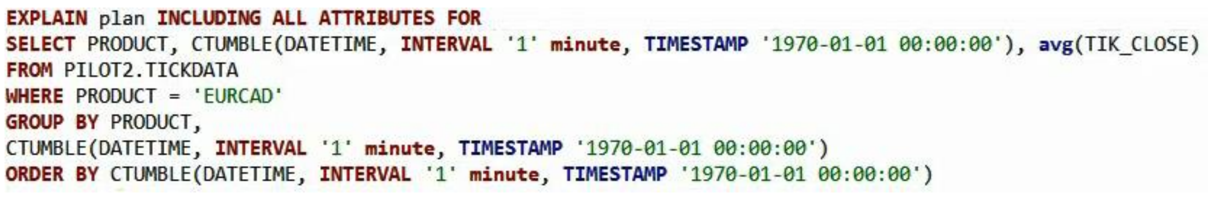
Here we have a standard SQL statement that is requesting the average value of the *TIK_CLOSE* column, from the corresponding data table, grouped by the *product* static field and the dynamically created period of time, which is a per minute base. The columns involved in the GROUP BY clause are also part of the project defined in the SELECT statement, while we also included a filter operation defined by the WHERE clause to avoid the full scan of the table. As the *product* is the first column defined in the primary key, the filter will make use of the related index, so the scan operation to retrieve the intermediate dataset that will be pushed to the aggregate operation will have a logarithmic complexity, instead of a linear. As graphical user SQL clients tend to only show the first 100s of rows of the result, and to make things fair, we also added the ORDER BY clause, which will force the INFINISTORE query engine to firstly calculate all rows of the aggregations, and then to order them. That way, it will always calculate the entire returned result set.

We executed this statement in the *vanilla* infinistore first. It took approximately 8 seconds for this operation to return. Then we executed the same statement using our *Online Aggregates* implementation. The result was retrieved in less than 800 milliseconds, which is 10 times faster. This means that the latency was 10% of the one required by a *vanilla* deployment. Things can get even worse if we remove the filter operation defined in the WHERE clause. In our historical dataset, we have financial currencies of 4 different products during a specific period of 8 months. If we remove the WHERE clause, the query engine has to traverse the whole parent table. In other words, it will have to calculate the result of the aggregation of an input intermediate dataset that is 4 times bigger. As a result, the latency of this execution with the *vanilla* datastore is around 30secs, while using the *online aggregates* is almost 2 secs.

What must be highlighted at this point, is that the *vanilla* datastore will have to traverse the whole data table each time an operation that implements aggregates is involved, while using the *online aggregates*, it will only have to traverse the data rows that are stored in the derived table. These rows depend on the granularity of the data that are involved in the GROUP BY clause: that is the primary key of the derived *delta table.*

In order for the reader to fully understand this concept, we will execute the same SQL statement, this time getting the average values of the *TIK_CLOSE* column on a per daily basis. We only need to change the reserved word *minute* to *day* in our CTUMBLE function. We need to remember now that we have initially defined another *online aggregate* that pre-calculates on a per daily basis. If we execute this statement in the *vanilla* datastore, the result will be the same: around 30secs. This is due to the fact that the relational query engine will always have to traverse the whole data table to calculate these values, and its data size is the same each time. Now let's do the same using our *online aggregates*. A day contains 24*60=1440 minutes. This means that our derived table related to the execution of the *online aggregate* per day will contain 1440 times fewer data. As it will also have to traverse the whole derived table, this means that this time, it will have to traverse 1440 times fewer data rows. As a result, the overall latency this time is just a couple of milliseconds, with the majority of them related to query compilation and data transmission, rather than the query execution itself. This means that the overall latency of the *online aggregates* is related to the granularity of the primary key of the data table, wherein in some cases it can be just a few rows. As a result, the computational complexity of the query execution can be near O(1).

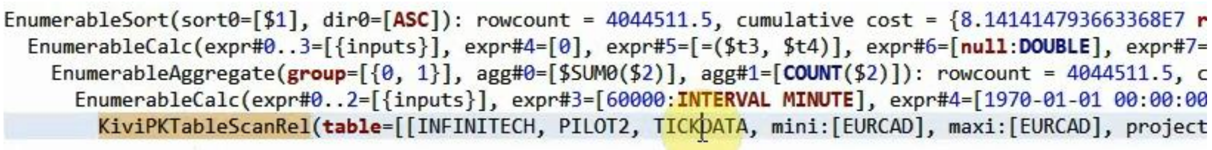
Let's see now how the relational query engine of the INFINISTORE internally works when it needs to execute *online aggregates.* In our example, we used the exact same standard SQL statement in both cases. To better understand how it works transparently to the data user, we need to investigate the query plans decided in both scenarios. To see the query plan of the query engine, we will execute the following statement, depicted in Figure 3.



```
EXPLAIN plan INCLUDING ALL ATTRIBUTES FOR
SELECT PRODUCT, CTUMBLE(DATETIME, INTERVAL '1' minute, TIMESTAMP '1970-01-01 00:00:00'), avg(TIK_CLOSE)
FROM PILOT2.TICKDATA
WHERE PRODUCT = 'EURCAD'
GROUP BY PRODUCT,
CTUMBLE(DATETIME, INTERVAL '1' minute, TIMESTAMP '1970-01-01 00:00:00')
ORDER BY CTUMBLE(DATETIME, INTERVAL '1' minute, TIMESTAMP '1970-01-01 00:00:00')
```

Figure 3: Asking for the query plan of the online aggregate

Using the *EXPLAN PLAN* reserved words, followed by the query statement that we initially submitted, the query engine will not execute the query plan, rather than returning to use the query plan itself, to investigate what would have decided to execute. For the vanilla instance, the query plan that has been decided is depicted in Figure 4.

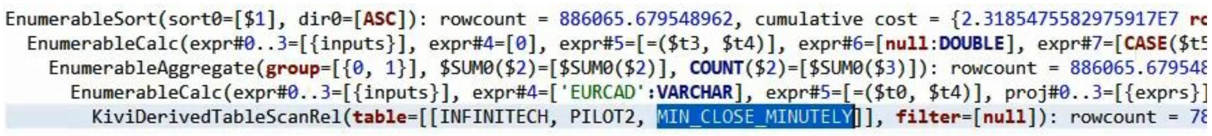


```
EnumerableSort(sort0=[$1], dir0=[ASC]): rowcount = 4044511.5, cumulative cost = {8.141414793663368E7 r
  EnumerableCalc(expr#0..3=[{inputs}], expr#4=[0], expr#5=[=($t3, $t4)], expr#6=[null:DOUBLE], expr#7=
    EnumerableAggregate(group=[{0, 1}], agg#0=[$SUM0($2)], agg#1=[COUNT($2)]): rowcount = 4044511.5, c
      EnumerableCalc(expr#0..2=[{inputs}], expr#3=[60000:INTERVAL MINUTE], expr#4=[1970-01-01 00:00:00
        KiviPKTableScanRel(table=[[INFINITECH, PILOT2, TICKDATA, mini:[EURCAD], maxi:[EURCAD], project
```

Figure 4: Query plan for vanilla datastore

A query plan is a graph of query operations that the output of each one is being used as an input by its upper layer in the graph. Giving analytical details on what this plan is actually doing is out of the scope of this report. However, what we can see from the query plan of Figure 4 is that at the bottom of the graph lies a KiviPKTableScanRel operation, which performs a scan over the parent table taking advantage of the primary key. The reason for this is that we initially included a filter operation using the WHERE clause over a column that is the first part of the primary key. The KiviPKTableScanRel operation will *attack* the TICKDATA data table that contains the raw data of the historical dataset and will have to traverse a vast amount of data with a computational complexity of O(log(n)), as we explained previously.

Let's now see the query plan that has been decided when having declared our *online aggregates*. The query plan can be depicted in Figure 5.



```
EnumerableSort(sort0=[$1], dir0=[ASC]): rowcount = 886065.679548962, cumulative cost = {2.3185475582975917E7 rc
  EnumerableCalc(expr#0..3=[{inputs}], expr#4=[0], expr#5=[=($t3, $t4)], expr#6=[null:DOUBLE], expr#7=[CASE($t
    EnumerableAggregate(group=[{0, 1}], $SUM0($2)=[$SUM0($2)], COUNT($2)=[$SUM0($3)]): rowcount = 886065.67954
      EnumerableCalc(expr#0..3=[{inputs}], expr#4=['EURCAD':VARCHAR], expr#5=[=($t0, $t4)], proj#0..3=[{exprs}]
        KiviDerivedTableScanRel(table=[[INFINITECH, PILOT2, MIN_CLOSE_MINUTELY]], filter=[null]): rowcount = 7
```

Figure 5: Query plan using online aggregates

We need to take a look at the operation of the bottom layer of the graph. In this case, the query engine identifies that has been already declared an *online aggregate* that is related to the submitted SQL statement, and instead of a scan on the *parent* table, it will make use of the KiviDerivedTableScanRel operation. The latter will scan the derived table called MIN_CLOSE_MINUTELY, which is actually the name of the *online aggregate* that we defined in the previous subsection and can be shown in Figure 1. A full scan in the derived table requires the traverse of a limited amount of data records that are related to the granularity of the primary key and can reduce the overall execution time from 30secs to just a couple of milliseconds, as we showed previously.

Concluding this chapter, what we have accomplished during the second phase of the project is to develop a framework that allows for defining the *online aggregates* in a declarative manner, and make use of them by executing standard SQL statements using standard JDBC connections. The relational query engine has been now integrated with this functionality that was provided by the storage layer during the first phase of the project, and now can redirect such analytics over the derived tables, transparently to the data user or application developer.

6 Performance Evaluation

In the previous sections, we firstly saw the basic principles behind the concept of the *online aggregates* and how our semantic concurrency control can guarantee transactional semantics and data consistency on both parent and derived tables, as data is being ingested. Then we demonstrated the use of both our direct API to the data storage engine, and how to make use of the *online aggregates* using standard SQL statements and let the relation query engine of the INFINISTORE create the corresponding query execution plan. The acceleration that is achieved when the query execution plan is targeting the derived tables is huge, due to the downgrade of the computational complexity of the operation itself.

As the previous section validated our performance acceleration when reading data, in this section the focus is given on evaluating the performance of our implementation when also ingesting data at very high rates. One of the most important outcomes of task T3.1 ("Framework for Seamless Data Management and HTAP") is the ability of the INFINISTORE to provide hybrid transactional and analytical processing. This is achieved by its dual SQL/NoSQL interface and the newly developed internal data structures of the data storage. The addition of the *online aggregates* extended these data structures, while on the same time, it forces the storage engine to perform additional write operations for a single insert, in order to both add the newly arrived data record to the parent table and also to calculate and consistently store the aggregate value. As a result, an additional overhead is expected to appear during the write operations, and this section provides the performance evaluation of our implementation. We will firstly start with measuring the performance of a *vanilla* data schema with no *online aggregates,* in order to give some insights on how the storage can behave, and then we will use 4 different examples defining *online aggregate,* and we will comment on the benefits and drawbacks of each decision.

For this evaluation, we relied on a real-life scenario that is currently being evaluated in one of the Proof-of-Concepts of LeanXcale with one of its clients. This proves that the outcomes of this task T5.3 have been already merged into the company's main product and have started being exploited in production. In this scenario, large enterprises need to use of the *data offloading* paradigm, and send data to the LeanXcale datastore in order to perform analytics there. The data being sent is being ingested at very high rates, exploiting the innovation that the INFINITORE can provide. The data is related with the daily finance transactions of credit cards from one of the major banks of Portugal. The schema of the data records being send is depicted in the following code snippet.

```
CREATE TABLE TRANSACTIONS (
    ID BIGINT,
    CREDIT_CARD VARCHAR(24),
    TYPE VARCHAR(1),
    QUANTITY DOUBLE,
    EXECUTION_DATE TIMESTAMP,
    PRIMARY KEY (CREDIT_CARD, EXECUTION_DATE, ID));
```

The data consists of a primary key, the number of the credit card, the type of the transaction, along with the amount of data being involved in this finance transaction, and the point in time when this transaction took place.

For the evaluation, we relied on the INFINITECH way of deployments using our prepared recipes for the deployment of the INFINISTORE, as demonstrated in the previous section. We deployed the INFINISTORE at the NOVA common testbed, using an installation of 4 CPU cores with 8GB of RAM. We also created a client application that will take as an input a csv file of 30.000.000 records and will start ingesting data to the INFINISTORE. The client application was also deployed in the NOVA cluster, in order to remove any additional overhead related with the network, trying to simulate a scenario close to reality. We used a container of 4 CPU cores and 8GB of RAM.

For the deployment of the INFINISTORE, we used 3 different types: one with a single data node, another using 2 data nodes and a third one having 4 data nodes. The data table was firstly split using a *hash* *distribution* over the available nodes and then we repeat the experiment with the data split but also using the bi-dimensional partition capabilities of the INFINISTORE, which provides additional acceleration during ingestion time in cases a column that is part of the primary key is type of TIMESTAMP. In particular, this allows the data storage engine to use 2 dimensions for indexing, and re-partition accordingly. As a result, as data is being ingested over a time period, data will be partitioned per time unit and will be removed from the in-memory indexes, as it will be very rare to be needed again. As the maximum size of a VM available to us in the NOVA cluster had 6 CPU cores, we could only scale the datastore to 4 data nodes (as 8 data nodes would have exceeded the number of the available cores, and as a result, it would have removed any benefit from distributed the dataset further).

For the use of the client applications, we performed the experimentation having 1 to 16 separate threads that are ingesting data to the INFINISTORE. The parallelism of the ingestion can be beneficial compared to the number of data nodes deployed in the INFINISTORE. In practice, the optimum option is to use 2**nodes* threads. For example, when having 2 data nodes, then the best option would have been to make use of 4 parallel threads for data ingestion. Having more threads ingesting data will not be beneficial, as they will have to compete for in the data node, and thus, one will block the other. This will be verified in the following subsection with the results of our evaluation.

Before starting our experimentation, it is very important to properly define the primary key of our schema and to wisely select the type of partitioning and the data fields. As mentioned before, the data load consists of various records regarding finance transactions of credit cards. The *ID* value that originally serves as the primary key will result in a key with very high granularity, so we will make use of the *CREDIT_CARD* along with the timestamp of the transaction. In order to cope with the rare situation that we have two financial transaction taking place at exactly the same time (or maybe, the TIMESTAMP value is *casted* into a DATE type), we will use the *ID* as the third column in the compound primary key. We also rely on the *hash partition* on the CREDIT_CARD column, as the distribution of the number of the credit cards is not known a priori, so that we could have chosen a partitioning based on the range of these values. An important drawback of the *hash partitioning* is that it does not give an important acceleration in the cases of range queries (that will be the most common case when we need to execute aggregation operations). However, we will rely on our *online aggregates* for this, so this is not a problem in our case. To make use of the *hash partitioning*, the DDL for the definition of our table will be slightly changed as follows:

```
CREATE TABLE TRANSACTIONS (
    ID BIGINT,
    CREDIT_CARD VARCHAR(24),
    TYPE VARCHAR(1),
    QUANTITY DOUBLE,
    EXECUTION_DATE TIMESTAMP,
    PRIMARY KEY (CREDIT_CARD, EXECUTION_DATE, ID),
    HASH(CREDIT_CARD) TO DISTRIBUTE);
```

When using our bi-dimensional partitioning, this will rely on the *EXECUTION_DATE* column, which is part of our primary key, as defined above. To instruct the storage engine to make use of this novel type of partitioning, introduced under the scope of T3.1, then the DDL will be slightly changed as follows:

```
CREATE TABLE TRANSACTIONS (
    ID BIGINT,
    CREDIT_CARD VARCHAR(24),
    TYPE VARCHAR(1),
    QUANTITY DOUBLE,
    EXECUTION_DATE TIMESTAMP AUTOSPLIT 'AUTO',
    PRIMARY KEY (CREDIT_CARD, EXECUTION_DATE, ID),
    HASH(CREDIT_CARD) TO DISTRIBUTE);
```

The next subsection will now provide the performance evaluation of our solution, using the setup described above. We will firstly start with a data schema with no *online aggregates,* and then we will compare and evaluate the performance overhead using the following scenarios:

- Aggregation grouped by the credit number
- Aggregation on the overall summary of values in the table
- Aggregation grouped by a single time period (per day)
- Aggregation grouped by 3 different time periods (per hour, per day, per week)

6.1 Ingestion with no online aggregates

We firstly deployed the INFINISTORE with no online aggregates, using the schema definition that was described above. We started with an instance using *hash partitioning* on a single data node, then having 2 data nodes and then with 4 data nodes. We repeated the experimentation having the *bi-dimensional partitioning* on 1, 2 and 4 data nodes. For each scenario, the client application sent 30.000.000 records for a time period of 60 days with a unique distribution of credit cards over a size of 100.000 different numbers of cards. We started with 1 thread generating data traffic, and then we increased to 2, 4, 8 and 16 threads. We repeated each benchmark 3-5 times and we kept the average of the latency that was witnessed for adding all the 30.000.000 records. Having the overall response time and the number of records that was added, we were able to calculate the throughput that the INFINISTORE could achieved, removing any additional network overhead due to the fact that both the INFINISTORE and the client application was deployed in the same local network inside the NOVA cluster. Figure 6 and Figure 7 shows the results with or without the use of the bi-dimensional partitioning. On the X axis we can see the number of the threads ingesting data, and in the Y axis the achieved throughput in records per second.

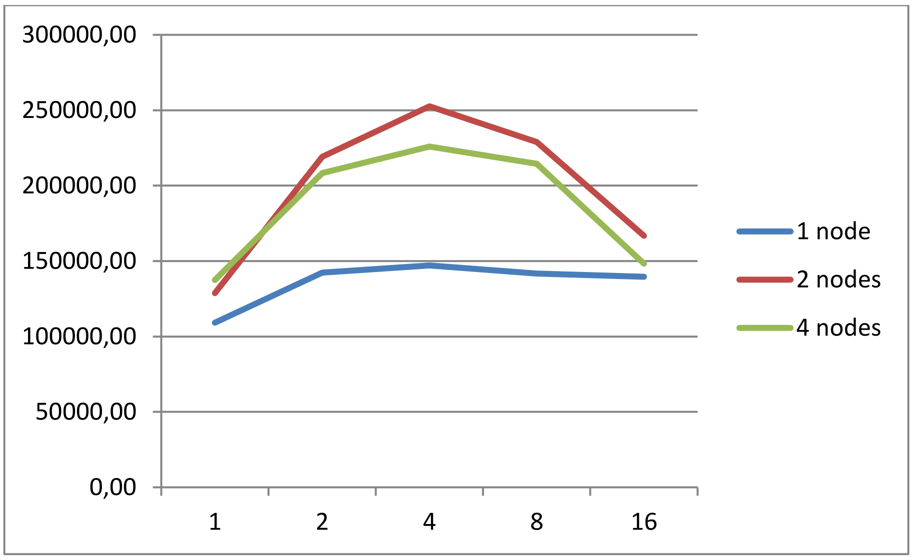


Figure 6: Hash-Join, no bi-dimensional, no aggregates

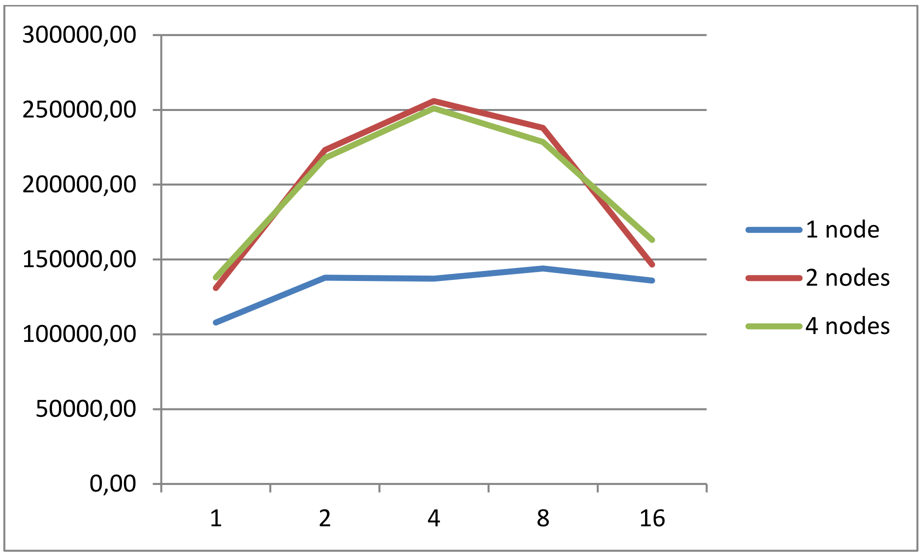


Figure 7: Hash-Join, bi-dimensional, no aggregates

As depicted in both figures, the maximum throughput that we observed was ~250.000 records per second in both cases, with a slight improvement when using our *bi-dimensional* partitioning, while it showed that the INFINISTORE can scale adequately and almost linearly up to 4 threads when doubling the number of the data nodes that the dataset will be partitioned. In the cases having a single node, we reached the limit very soon, and even if we increase the input load having 2 to 16 threads generating traffic, our limitation comes from the bottleneck in the storage engine. A single data node cannot serve all incoming data load and therefore it can only reach the maximum throughput that can be achieved by the node itself. Putting an additional node, the throughout is two times faster, especially when we use 4 threads, applying the rule of thumb *threads = two time the number of data nodes*. With 4 data nodes, things should have been increased even further, but at that point, we reached the limitation of the cluster itself that could not provide us with more CPU cores to take the full advantage of the storage engine. That's the reason also why the throughput reaches its maximum value around 4-8 threads, as increasing the data load even further with 16 nodes would not be beneficial as we only can use up to 4 data nodes.

Having this first round of experimentation for the performance evaluation of our implementation, we now have the basis to compare the additional overhead that is expected to be observed when using different types of *online aggregates*.

6.2 Ingestion with aggregation of very high granularity

We will now provide an example on how the data user or application developer may make use of the *online aggregates* but with no or very minimum benefit. This is a demonstration of a scenario when the data user should not use this mechanism. In our example, we will define an *online aggregate* to automatically calculate for us the summary of the money involved in the transactions that belong to a credit number. This means, all money spent for a particular credit card. The definition of such *online aggregate* can be depicted in the following code snippet:

```
CREATE ONLINE AGGREGATE SUM_QUANITY AS
sum(QUANTITY) SUM_QUANITY
FROM TRANSACTIONS
GROUP BY CREDIT_CARD;
```

We repeated the same experiment as before. We used exactly the same configuration, with the same deployments, having the same client. We only defined additionally the *online aggregate* and then we started with a deployment with 1 data node, and we scaled it out to 2 and 4 data nodes, with or without *bi-dimensional* partitioning, using a client application generating data traffic having 1 to 16 threads. The results are depicted in Figure 8 and Figure 9.

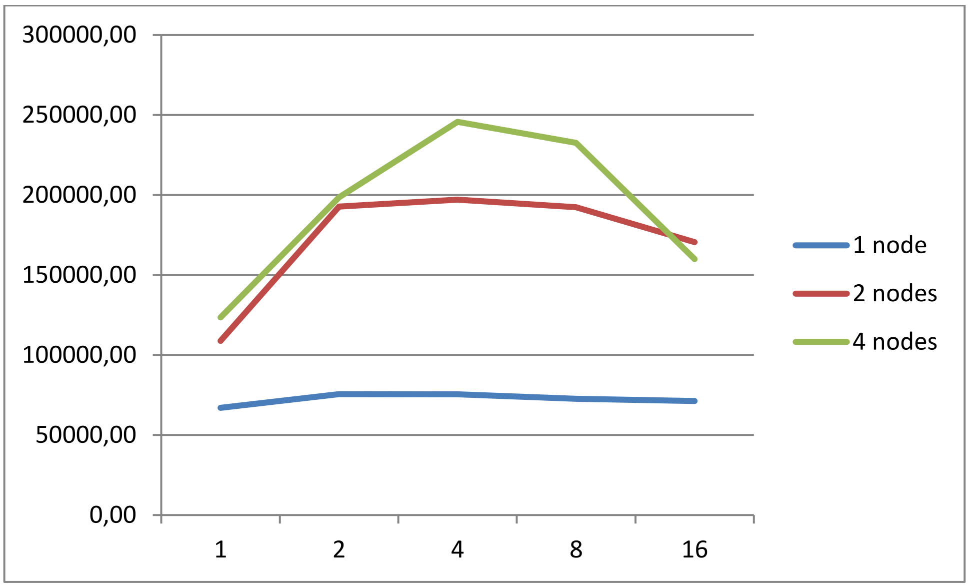


Figure 8: Hash-Join, no bi-dimensional, aggregation with high granularity

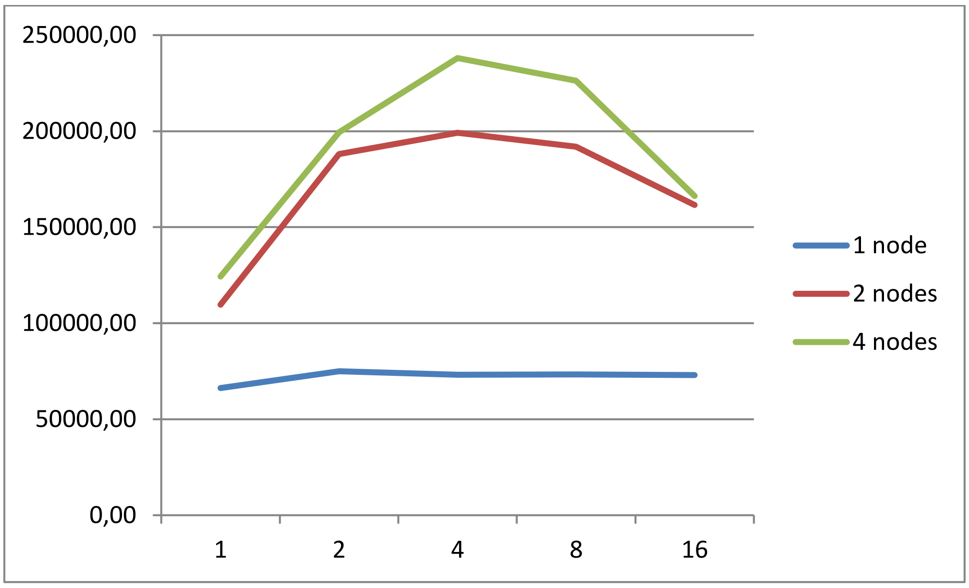


Figure 9: Hash-Join, bi-dimensional, aggregation with high granularity

What we can conclude from these figures is that they follow the same pattern, as the ones without online aggregates. This will be the case in all following benchmark experimentation, which proves that our solution can scale as affectively as without *online aggregates*, and they face the similar behaviour. For instance, having a single data node, the solution cannot perform a better throughput when increasing the number of the ingested data traffic and reaches its limitation very soon. Also, doubling the nodes, the performance is also 2 times better, until reaching the size of 4 data nodes, which equals the number of CPU cores provided to the deployment.

What can be also concluded is that we have a similar performance in both a vanilla data schema and one with an *online aggregate*, but slightly lower in the latter case. Let's see now how much overhead our newly developed *online aggregate* has added. Figure 10 and Figure 11 shows the percentage of the performance overheads.

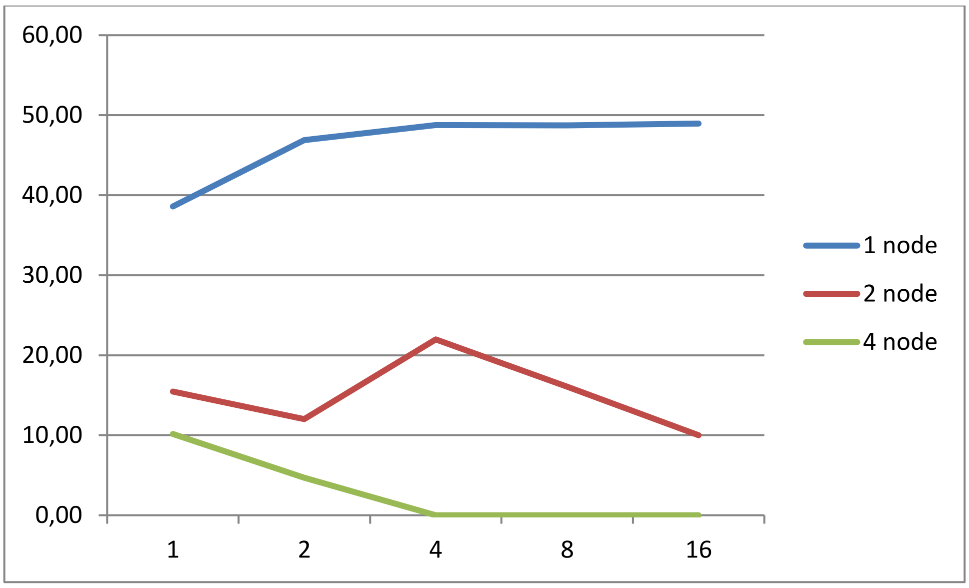


Figure 10: Percentage overhead, hash-Join, no bi-dimensional, aggregation with high granularity

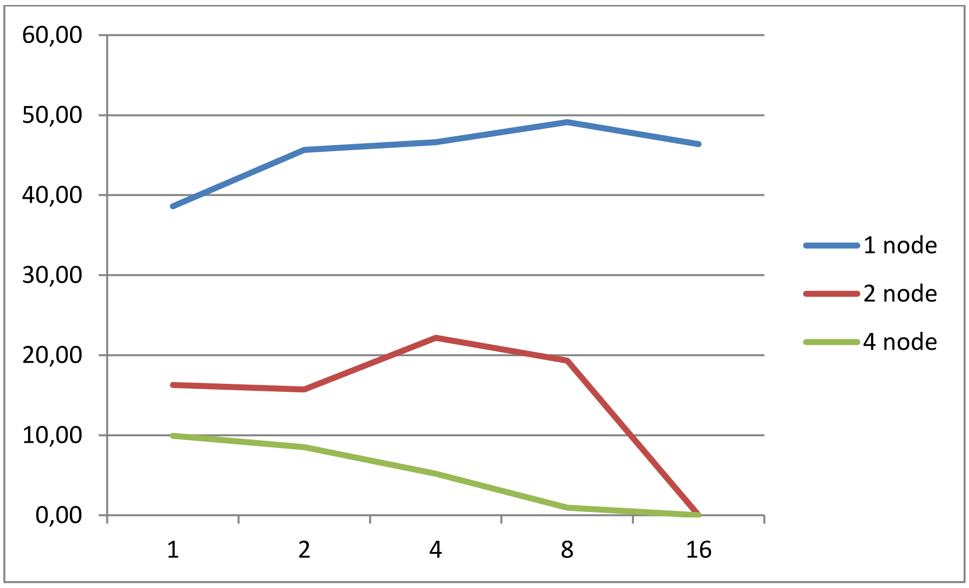


Figure 11: Percentage overhead, hash-Join, bi-dimensional, aggregation with high granularity

Firstly, we need to highlight here that our *online aggregate* is based on a column with very high granularity. We have 100.000 distinct credit card numbers, which creates a granularity of 100.000 distinct values in the internal derived table, which is adequately high. This is depicted especially in the case when we have 1 single node, and all insertions are targeting the same node. This data node reaches its limit very soon and by adding an *online aggregate* of such high granularity, the overhead is up to ~50%. When having additional nodes, the workload is split among the tables, so we can notice an overhead of ~20% in case of 2 nodes and ~10% in the case of 4 nodes. When increasing the number of threads that produce incoming data traffic, the overhead is minimum, but this is due to the fact that the INFINISTORE is under-performing in both the vanilla scenario and the one with the *online aggregates* as a result of the saturation of the resources consumed by the INFINISTORE. To put it simpler, the INFINISTORE's observed throughput was also bad in the vanilla scenario and we would have needed to create an additional instance in the NOVA cluster, thus increasing the number of data nodes to 8.

Let's see now the benefit with regards to performance acceleration when using *online aggregates* or not. Now, the data user wants to see the overall (summary) money spent for a specific credit card. Using the *online aggregate*, the data user would need to execute the following standard SQL statement (which *attacks* the *SUM_QUANTITY* derived table):

```
SELECT * FROM SUM_QUANITY
WHERE CREDIT_CARD = '6771-8900-0000-1220
```

The query was executed from a local machine, and thus includes an overhead from the network latency. The average response time of such an execution, including the network latency is around 300msecs.

If the data user is using the *TRANSACTIONS* data table, so by-passing the online aggregates, then the standard SQL statement will be the following:

```
SELECT sum(QUANTITY) FROM SUM_TRANSACTIONS
WHERE CREDIT_CARD = '6771-8900-0000-1220
GROUP BY CREDIT_CARD
```

We executed the aforementioned query from our local machine several times, and the average latency was around 310msecs. This implies that we have no important performance acceleration given the fact that we spent ~20% overhead when ingesting the data.

Let's take a close look now why this is happening. The query execution plan when *attacking* directly the derived table which has the *online aggregate* is depicted in the following snippet:

```
KiviDerivedTableRangeScanRel(
table=[[SARGA, APP, SUM_QUANITY]],
aggregateQuery=[SUM($SUM_QUANTITY) group by $CREDIT_CARD],
min=[[6771-8900-0000-1220]], includeMin=[true], max=[[6771-8900-0000-1220]], includeMax=[true])
```

This query plan defines a scan operation in the derived table SUM_QUANTITY, with the min/max number of the value of the credit card. This is what it was expected. The query optimizer selects the plan that will *attack* this derived table with the pre-calculated value. But what happens when we use the standard SQL without explicitly defined the *online aggregate.* The query execution plan is depicted in the following snippet:

```
KiviAggregateTableScanRel(
table=[[SARGA, APP, TRANSACTIONS, mini:[6771-8900-0000-1220], maxi:[6771-8900-0000-1220],
project:[1, 3],
aggregate_group:{0}, aggregates:[SUM($1)], project:[1]]])
```

Here, we can see that instead of *attacking* the derived table (as it should have been selected by the query optimizer), the execution plan will access the parent table that contains all row data. In fact, it will make use of the *KiviAggregateTableScan* operator of the relational query engine of the INFINISTORE, which actually pushes the aggregation down to the storage engine. The reason for this is that the storage will need to perform a *scan* operation on the parent table, however, the column to perform the *scan* is the first field of the primary key of that table. This means that it will go directly to the index, so it will be very fast. Moreover, as it will push down the aggregate operation, this will be calculated in the storage layer and not data will need to be transmitted to the query engine. Another important reason is the level of granularity of the column, which is very high and as a result, the storage would need to access very few items to calculate the overall summary of the money spent for a specific credit.

This scenario is an excellent example on when the use of *online aggregates* does not make much sense, and that is why it was included in this document. The following scenarios will demonstrate the use of the *online aggregates* in cases with very low granularity and a combination of different operations.

6.3 Ingestion with aggregation of very high data contention

In this scenario, we have the exact opposite use case: the data user now needs to calculate the overall money spent concerning the summary of the transactions that took place in the bank. The definition of this *online aggregate* is now depicted in the following code snippet:

```
CREATE ONLINE AGGREGATE SUM_QUANITY AS
sum(QUANTITY) SUM_QUANTITY
FROM TRANSACTIONS
```

Here we have the lowest level of granularity, which is 1, as all the pre-calculated data will be stored in a single row of the derived table. This happens because the data user wants to calculate the overall summary of the money spent and therefore there is no group by clause. On the contrary, this implies that the data storage engine will have to deal with a data congestion of very high rate, as all incoming transactions will need to update the same row. Therefore, the scope of this benchmark is to evaluate the performance of our novel *semantic concurrency control* mechanism that has been designed to remove the conflicts that normally would have occurred due to this high data congestion.

Figure 12 and Figure 13 depicts the performance observed with this type of *online aggregation,* with or without bi-dimensional partitioning where the important factor to observe is that even if we should have witnessed the effects of the high data contention when trying to update the same data record of the derived table, this does not create any significant overhead to the overall throughput that the INFINISTORE can accomplish. In both cases, the maximum is around 250.000 records per second when having 2 or 4 data nodes using 4 or 8 threads for generating incoming data traffic. The points when the performance starts to drop are the same and are due to the saturation of the resources of the machine. This proves that our *semantic concurrency control* that has been implemented works very efficiently and in fact, can now combine the benefits of ingesting data at very high rates, but also removing both the race conditions of the NoSQL world and the data contention due to write-write conflicts of the SQL transactional world.

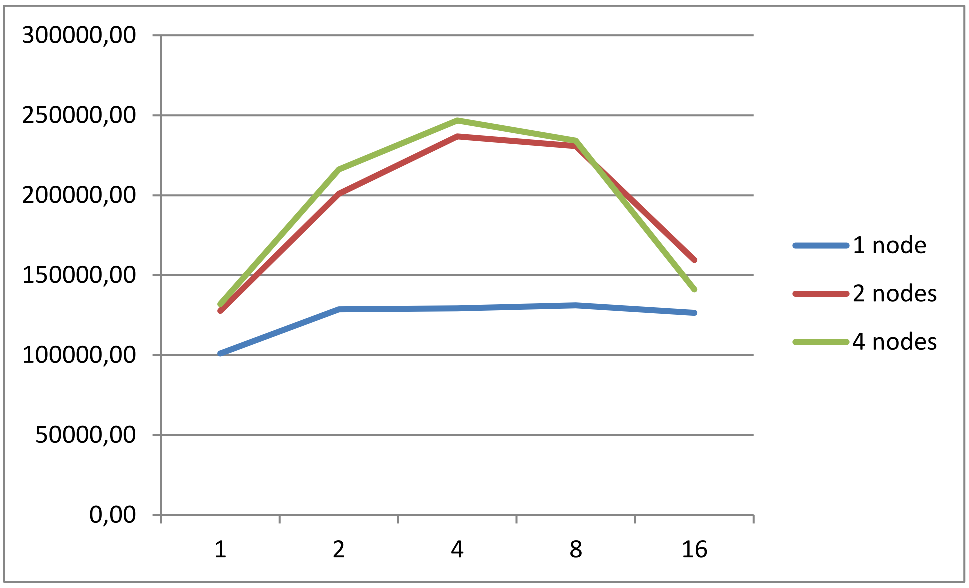


Figure 12: Hash-Join, no bi-dimensional, aggregation with high contention

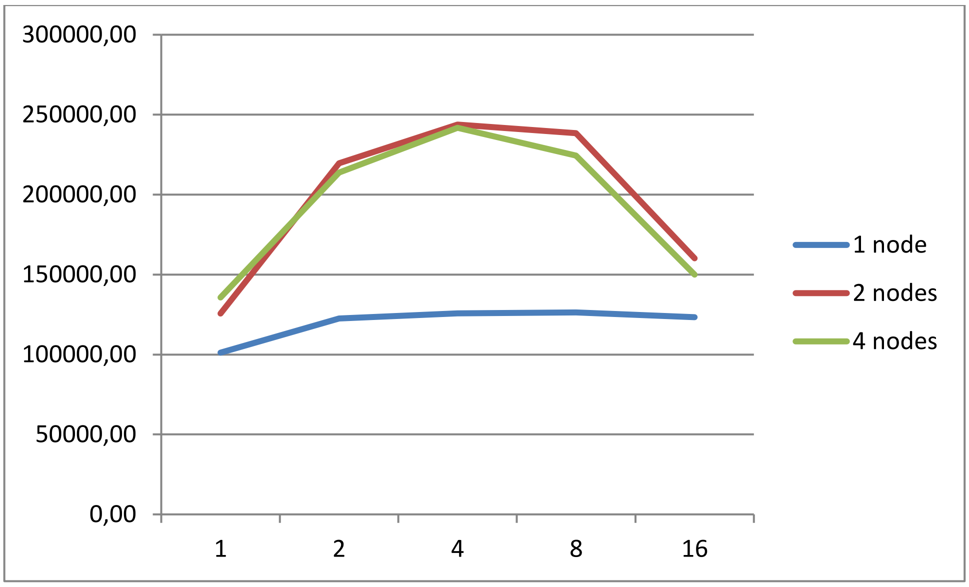


Figure 13: Hash-Join, bi-dimensional, aggregation with high contention

Figure 14 and Figure 15 depict the percentage of the performance overhead with this type of *online aggregate* with or without *bi-dimensional* partitioning. We can see that even in the cases where we used only 1 data node, the overhead is significant lower than the previous cases, while it drops even around 2% when using additional data nodes. The reason for that is that pre-calculation takes place in memory using our novel *semantic concurrency control* mechanism that is not affected when having huge data contention, created by the need of all incoming database transactions to access and update the same data row of the derived table. Therefore, the increased additional overhead that was observed in the previous example was related with the granularity of the column that was involved in the *group by* clause. In this second example, the granularity is just 1, and this effect is not observed.

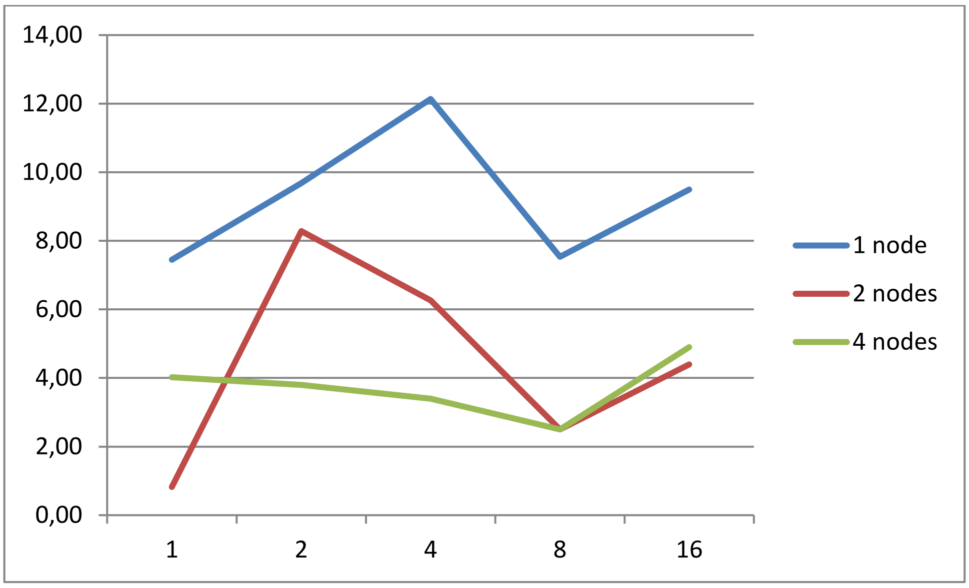


Figure 14: Percentage overhead, hash-Join, no bi-dimensional, aggregation with high contention

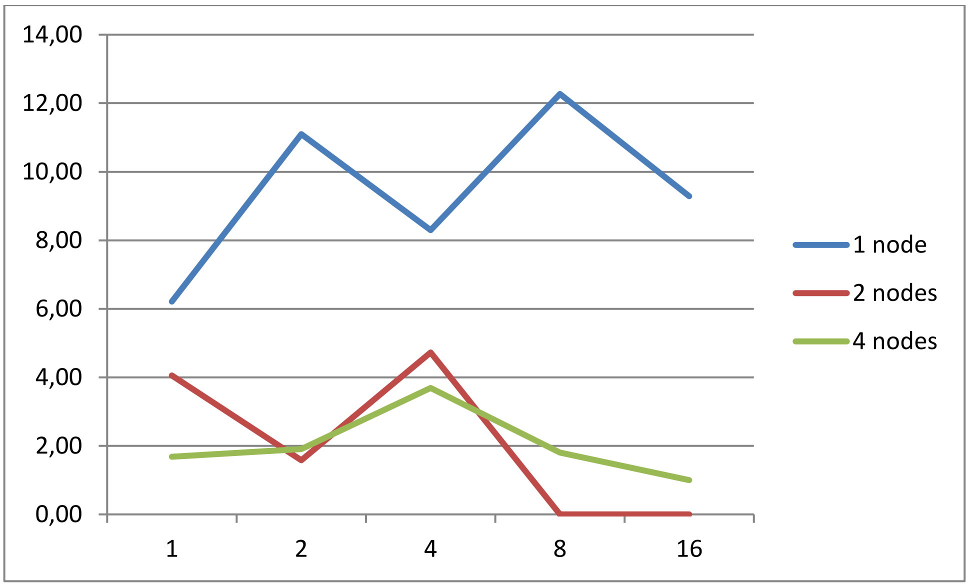


Figure 15: Percentage overhead, hash-Join, bi-dimensional, aggregation with high contention

Let's see now the difference of having or not defined an *online aggregate*. Without *online aggregate* we should have used the following standard SQL query statement:

```
SELECT sum(QUANTITY)
FROM TRANSACTIONS
```

The query execution plan that our query optimizer decided is the following:

```
KiviAggregateTableScanRel(
table=[[SARGA, APP, TRANSACTIONS, aggregate_group:{}, aggregates:[SUM($3)]]]):
rowcount = 1.0,
cumulative cost = {2.9914574E7 rows, 2.99145731E7 cpu, 0.0 io}, id = 51
```

Without *online aggregates,* we can see that the *KiviAggregateTableScan* operator is also selected, which makes sense as there is no derived table for the query optimizer to generate such a plan for. From the attributes of that operator it can be observed that a cumulative cost of 30.000.000 access is expected, which is related with the need to access all 30.000.000 records of the parent table. The average response time for executing this SQL statement is around ~10-12 secs. It is relatively low and this due to the newest improvements of the INFINISTORE, as developed within the scope of the task T3.1 and the ability to push down operations to the storage engine itself. Normally, the average response time would have been much greater as all records of the parent table would have been needed to be transmitted to the relational query engine for the latter to calculate the final value. This can be now down in the storage level, however, it still needs to access all 30.000.000 rows of that table.

When using the *online aggregates,* for the same SQL query statement, the query optimizer will now choose the following query execution plan:

```
KiviDerivedTableScanRel(
table=[[SARGA, APP, SUM_QUANITY]],
aggregateQuery=[SUM($SUM_QUANTITY)]): rowcount = 1.0,
cumulative cost = {1.0 rows, 1.0 cpu, 0.0 io}, id = 88
```

Here, the *KiviDerivedTableScan* has been selected that will *attack* the derived table, having a cumulative cost of 1, instead of 30.000.000 as before. This happens because the granularity of the derived table is just 1. As a result, the same query now responds in ~300milisecs instead of 10-12 seconds.

6.4 Ingestion with a single online aggregation

The remaining 2 subsections will evaluate the performance overhead when having *online aggregates* of a normal granularity per a specific time period. In this case the data user wants to check the average of money spent per hour, day or week. We will first define a single *online aggregate* and we validate the performance overhead and in the later section, we will define all three and see how the performance is affected when adding more.

For defining the *online aggregate* that calculates the overall daily money spent, the data user or application developer needs to execute the following statement:

```
CREATE ONLINE AGGREGATE SUM_COUNT_DAILY AS
sum(QUANTITY) QUANTITY_SUM,
count(QUANTITY) QUANTITY_COUNT
FROM TRANSACTIONS
GROUP BY
CTUMBLE(EXECUTION_DATE, INTERVAL '1' day, TIMESTAMP '1970-01-01 00:00:00') DAY_PERIOD;
```

As we have imported data for a time period of 60 days, the granularity of the derived table will be 60. Figure 16 and Figure 17 illustrates the performance observed when executing the same experiment using the same configurations as in the previous examples. Again we can notice the same trend as previously.

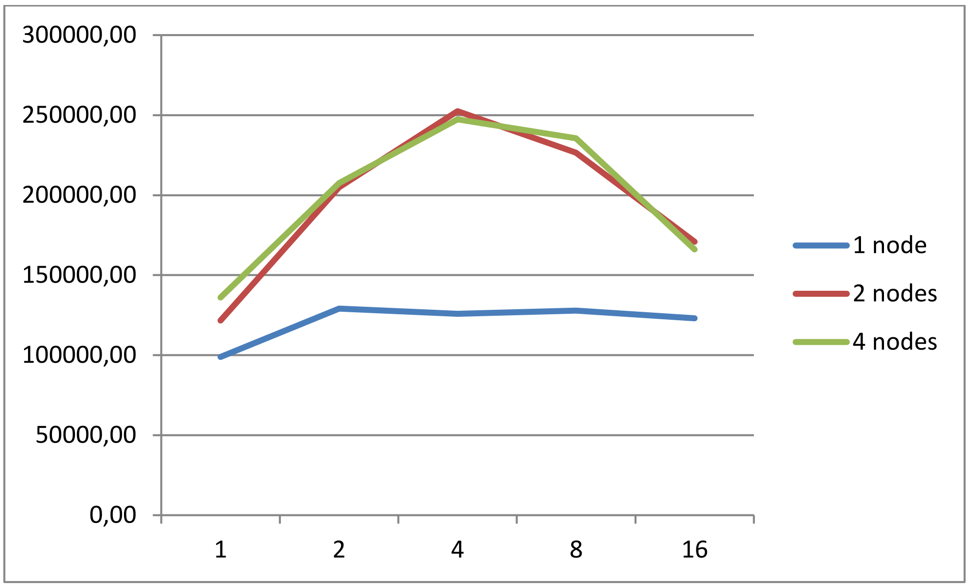


Figure 16: Hash-Join, no bi-dimensional, 1 online aggregation

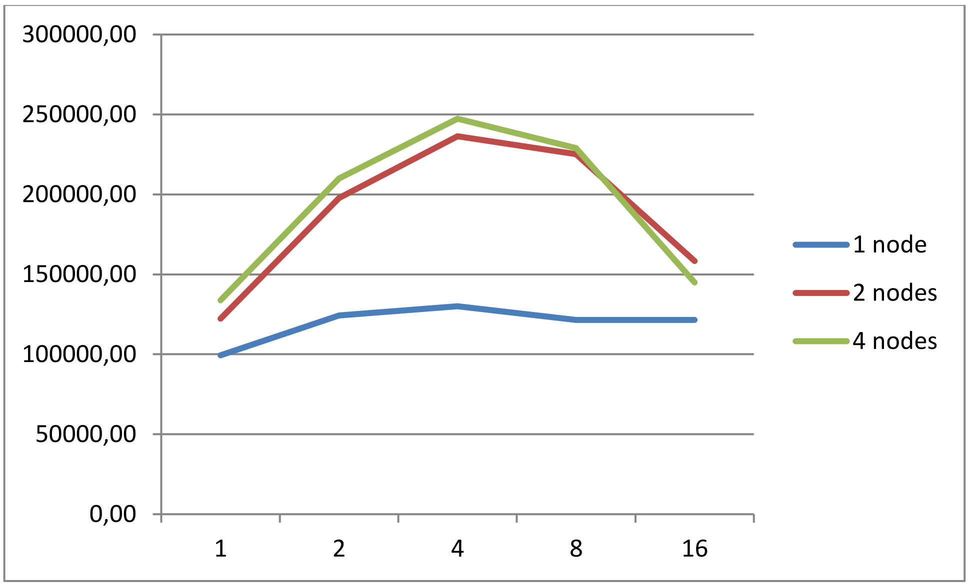


Figure 17: Hash-Join, bi-dimensional, 1 online aggregation

Regarding the percentage overhead of the performance when having *online aggregates* or not, Figure 18 and Figure 19 depicts these values. As before, when adding more data nodes the overall execution is much better compared when using a single node where all ingested data will need to be stored in the same node. The overhead is below 5% when the deployment can efficient make use of the available resources. This means that it does not reach the limitation of the single node, neither would require more resources due to the fact that we deployed data nodes that cannot be served by the available resources or the number of threads that produce the incoming data traffic are much bigger than the threads that should have been opened connections to the datastore.

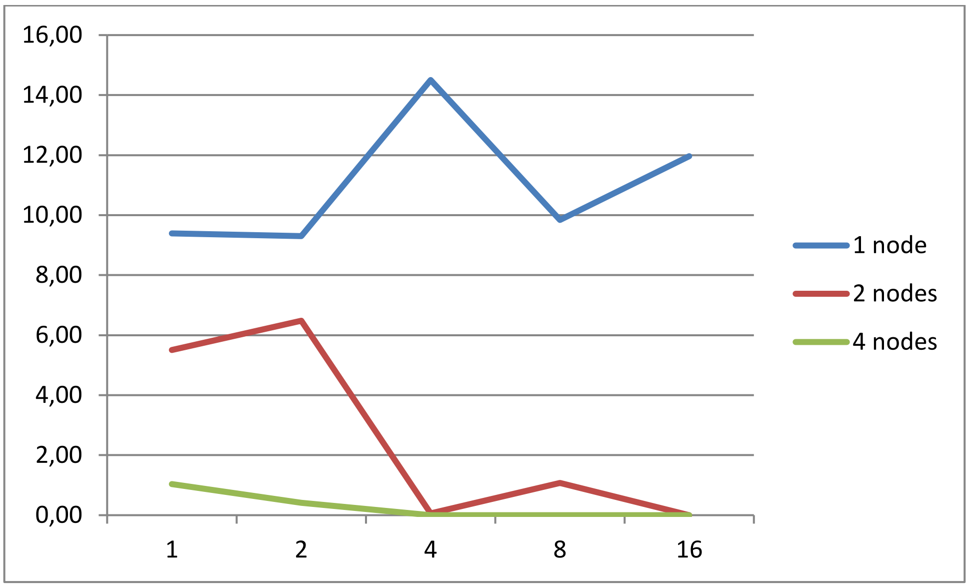


Figure 18: Percentage overhead, hash-Join, no bi-dimensional, 1 online aggregation

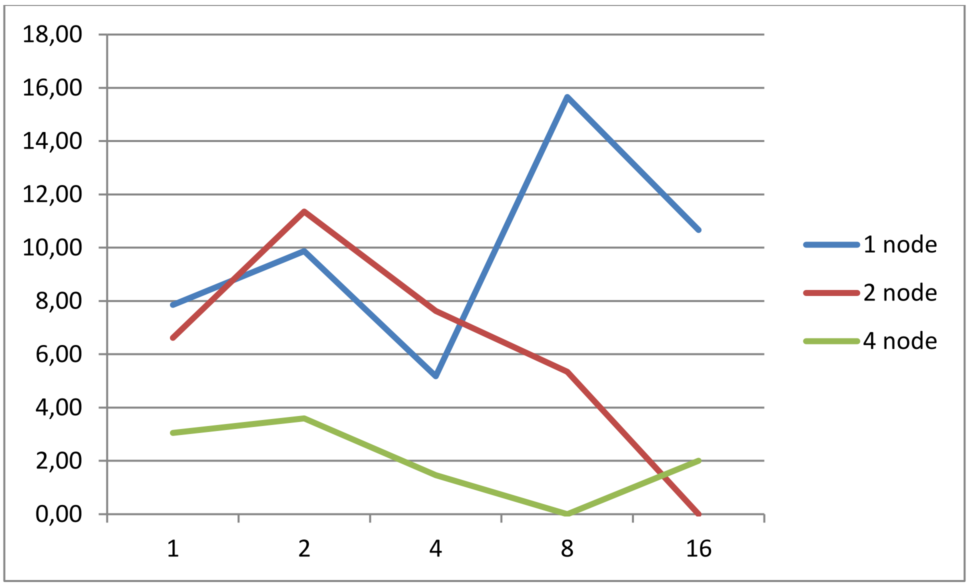


Figure 19: Percentage overhead, hash-Join, bi-dimensional, 1 online aggregation

6.5 Ingestion with three online aggregates

In our last scenario, the data user will define three *online aggregates* to calculate the summary of money spent per hour, day or week. The following statements will define these aggregations.

```
CREATE ONLINE AGGREGATE SUM_COUNT_HOURLY AS
sum(QUANTITY) QUANTITY_SUM,
count(QUANTITY) QUANTITY_COUNT
FROM TRANSACTIONS
GROUP BY
CTUMBLE(EXECUTION_DATE, INTERVAL '1' hour, TIMESTAMP '1970-01-01 00:00:00') HOUR_PERIOD;;
```

```
CREATE ONLINE AGGREGATE SUM_COUNT_DAILY AS
sum(QUANTITY) QUANTITY_SUM,
count(QUANTITY) QUANTITY_COUNT
FROM TRANSACTIONS
GROUP BY
CTUMBLE(EXECUTION_DATE, INTERVAL '1' day, TIMESTAMP '1970-01-01 00:00:00') DAY_PERIOD;


CREATE ONLINE AGGREGATE SUM_COUNT_WEEKLY AS
sum(QUANTITY) QUANTITY_SUM,
count(QUANTITY) QUANTITY_COUNT
FROM TRANSACTIONS
GROUP BY
CTUMBLE(EXECUTION_DATE, INTERVAL '7' day, TIMESTAMP '1970-01-01 00:00:00') WEEK_PERIOD;
```

Using the same configuration and methodology, the observed throughout put is depicted in Figure 20 and Figure 21. We can notice here a decrease of the performance in terms of throughput, which is normal as now the data storage needs to pre-calculate 3 aggregations instead of 1, for the same record that needs to be added. However, the overall throughput is still very high and can reach more than 200.000 records per second, exploited the innovation of the INFINISTORE as implemented under the scope of T3.1. Moreover, in cases we need to achieve even higher level of throughput, we can scale our deployment by adding an additional container having a couple of more nodes, or use an infrastructure that can create a virtual machine or container with more CPU cores, so that we can start the datastore with 8 or more data nodes. As a result, exploiting the provision of efficient scalability of the datastore, adding more nodes can allow us to achieve even higher throughput.

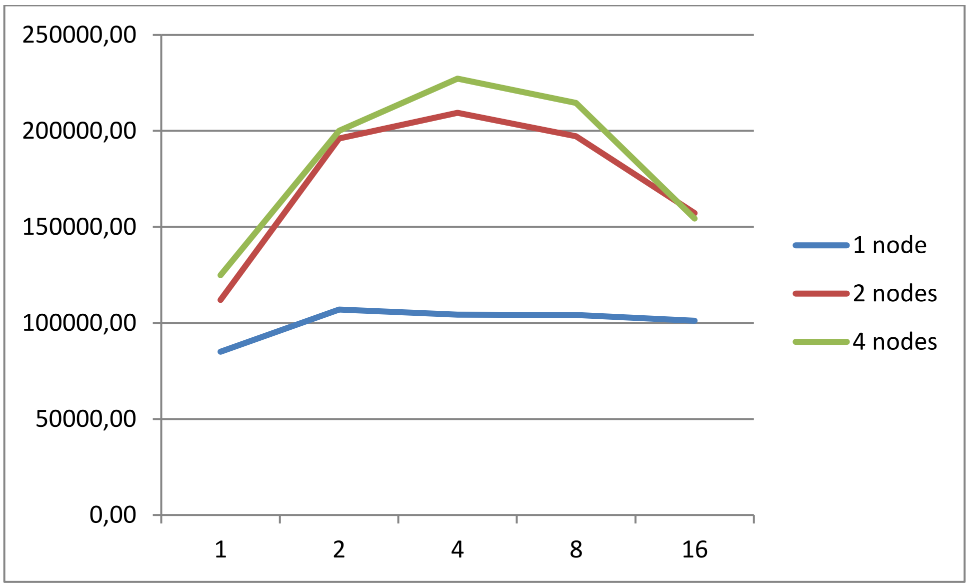


Figure 20: Hash-Join, no bi-dimensional, 2 online aggregations

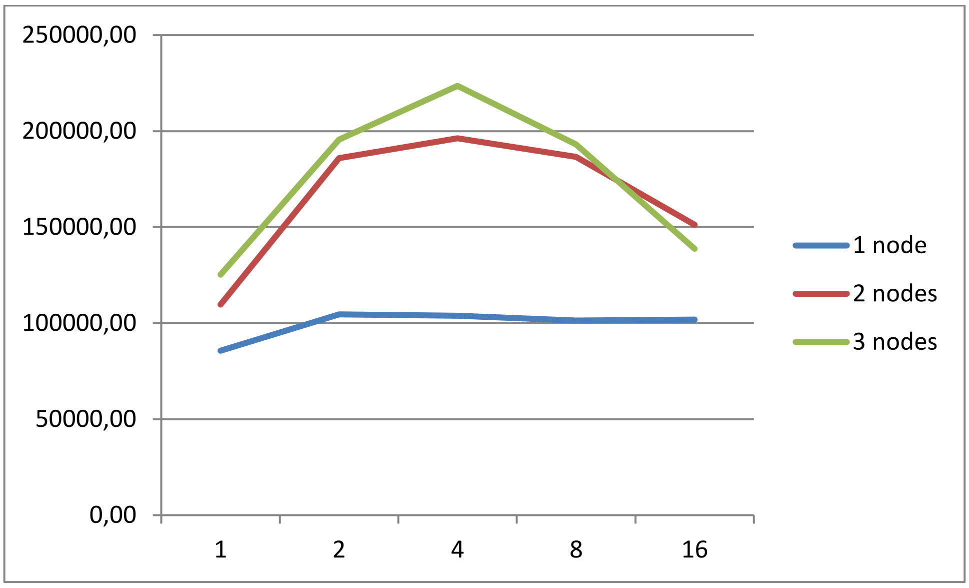


Figure 21: Hash-Join, bi-dimensional, 3 online aggregations

Regarding the percentage of the overhead that has been observed in this scenario, compared to the previous one, this can be depicted in Figure 22 and Figure 23. We can see from these figures that the overhead is around three times bigger than the overhead observed in the previous case. That is normal as in this latest scenario we have defined 3, instead of 1, *online aggregate*.

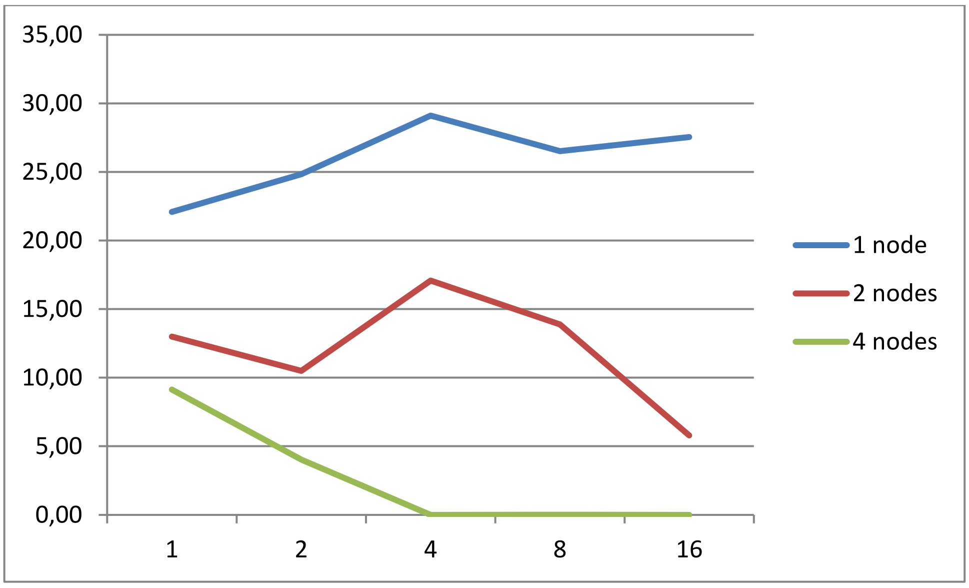


Figure 22: Percentage overhead, hash-Join, no bi-dimensional, 3 online aggregations

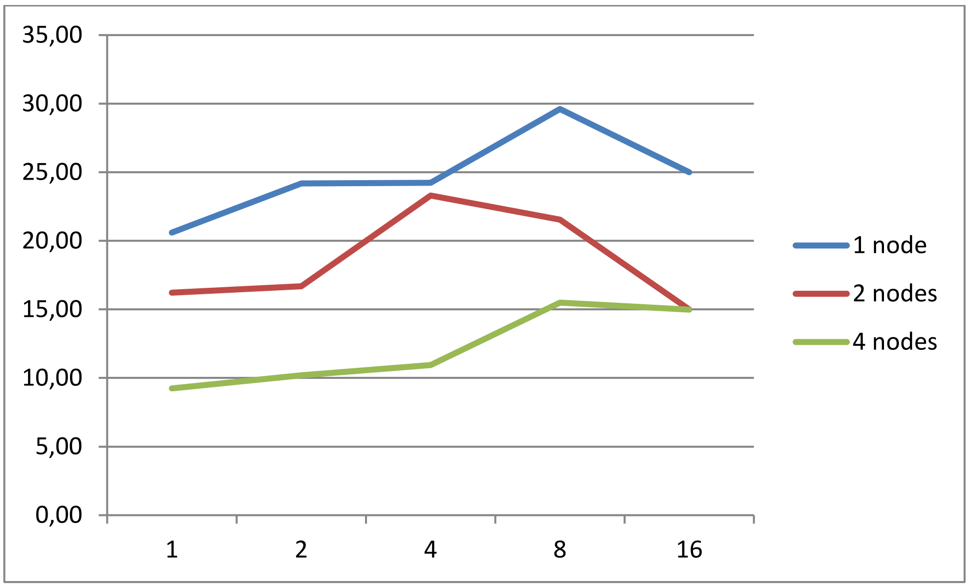


Figure 23: Percentage overhead, hash-Join, bi-dimensional, 3 online aggregations

7 Conclusions

This report documented the work that has been carried out in the scope of task T5.3 “Declarative Real-Time Data Analytics” whose main objective is to provide a framework that enables the execution of data analytics in real-time (online) in a declarative fashion. Towards this, we firstly identified the need for BigData applications in the financial and insurance sector to execute data analytic operations in real-time, while their datasets are being continuously updated by a data ingestion process at high rates. We presented the main technological barriers that are currently being tackled by the system architectures and what solutions exist and are commonly used. Then we investigated how we could benefit by pre-calculating the aggregated value beforehand and by storing this value in a separate field, inside a particular column of a data table. For this, we proposed the notion of *aggregate tables*. We verified the feasibility of our proposal when being used by both SQL and NoSQL database management systems, and we realized that both sides of these data ecosystems deal with significant issues: in the NoSQL world data consistency is broken due to the *Lost Updates* anomaly, while in SQL solutions, efficient execution of the statements is not feasible due to the high contention that the database transaction create. In order to overcome this, and exploit the hybrid transactional and analytical processing (HTAP) capabilities of the INFINITECH data management layer, we introduced the *online aggregates*. Those can in fact assist in overcoming the barriers explained before by relying on the semantic concurrency control mechanism of the database.

After introducing the *online aggregates*, we provided documentation on how these can be used by exploiting the *direct API* that allows data connection with the storage engine of the datastore. The important thing here is the definition of the *aggregated tables* which are needed to store the operational (delta) rows along with the tables themselves that hold the raw data. Additionally, each insert operation in a raw table will require the application developer or data scientist to also take care of the fact that he/she needs to insert the corresponding operational rows in the *aggregated tables*. Then, by executing a standard SQL statement, the query engine can transform it and take benefit from the additional structures that contain the pre-calculated values. Moreover, a hands-on demonstrator was presented that can be used as a guideline for application developers and data scientists on how to integrate their solutions with our framework. This has been validated by pilot#2 (“Real-time risk assessment in Investment Banking”) of the project, that makes extensive use of our approach. We have shown how the data user or application developer can benefit from the use of the *online aggregates* and execute complex analytical operations in times of magnitude faster than traditional approaches, ensuring data consistency at the same time and avoiding high contention during the data ingestion process. This shows how our solution helps developers as well. In order to have online analytics over an operational dataset, the developers will have to either rely on one of the existing database technologies, or go to a hybrid approach. With the latter, apart from being very expensive to maintain, they will have to sacrifice the online aspect. If they decide to go to one of the existing technologies, this will have to sacrifice either data consistency (which is unacceptable in the finance domain) or concurrency (which cannot be done in BigData) and the cost for the developers to overcome these problems is very high.

Finally, at the last phase of the project we also did an extensive benchmark analysis of our implementation in order to evaluate the performance overhead that can be observed when ingesting data at very high rates having the *online aggregates* defined. For that, we relied on a real-life scenario from one of the newly established Proof-of-Concepts of LeanXcale with its potential customers. We firstly executed a highly intensive workload of 200,000s rows per second, without any *online aggregate*. We then provided different use cases: one having an *online aggregate* of very high granularity, where the benefits of using such a structure not significant, and then with an *online aggregate* of the lowest granularity where the effect of very high data contention appears as hundreds of thousands of transactions try to update the value of the same record in each second. We proved that our novel *semantic multi-version concurrency* mechanism is highly effective and the observed overhead during data ingestion is significant low, especially compared to the performance acceleration of the read operations. Finally, we used two more examples, one with an *online aggregate* of normal granularity, and another using three *online aggregates* so evaluate how the performance is affected with the number of such defined data structures.

To conclude, the task T5.3 can be considered successful, as all its objectives have been addressed. Moreover, the technology that was created under the scope of this task is already filed for a patent and all the implementation has been already merged into the LeanXcale's main product, its datastore which is the base of the INFINISTORE - the data management layer of the INFINITECH project. Having that, it can deployed and installed using the INFINITECH way of deployments. Last but not least, the innovation created under the scope of this task, already incorporated in LeanXcale main product is being currently used by its client and have created new paths for exploitation in areas where the main pain is the ability to effectively execute analytical queries while data is being ingested at very high rates at the same time. This combines the outcomes of T3.1 with the technology implemented in this task. Currently, new Proof-of-Concepts have been established with potential clients of LeanXcale in order to further validate this technology and put it into production.

Table 4: Conclusions (TASK Objectives with Deliverable achievements)

Objectives	**Comment**
extend SQL in order to support the declaration, execution and configuration of queries over incremental algorithms	SQL extensions have been introduced that allows the data users to declare the *online aggregates* data structures to be used for the execution of the analytical queries that have pre-calculated the result. A demonstration is included in this deliverable providing examples for its usage.
extended SQL operations will create new analytics columns that will facilitate the deployment of full-fledged distributed incremental analytics algorithms directly connected to the operational database	The *online aggregates* create a novel data structure as a derived table of the parent one that contains the operational data and introducing new types of columns, named *delta columns*. New SQL operations have been implemented in the relational query engine that can target these derived tables and the *query planner* can take into consideration these new developed operations to generate a query execution plan that will benefit of the novel structures. The parent table contains the operational data of the datastore while the outcomes of this task in combination with the outcomes of the task T5.2 allows for incremental analytics using our novel data structures.
results of the algorithm will be also exposed as values in columns of the indicated table of the database	The results are being pre-calculated during runtime, as data is being continuously ingested, ensuring data consistency and the results are stored as values in the aforementioned *delta* columns of the derived tables
once the infrastructure for supporting declarative analytics will be developed, the data scientist will be able to invoke it as a standard SQL feature of the database	Once the *online aggregates* have been defined, the data user can submit standard SQL statements to the database. The query planner will make use of the novel SQL operations and will redirect the execution of the query statement to the newly developed data structures. As a result, the data user executes standard SQL statements that are being transparently executed in the derived tables. Concrete examples of query plans have been included in this deliverable that show how this happens in different scenarios.

Table 5: Conclusions – (map TASK KPI with Deliverable achievements)

KPI	**Comment**
Average reduction in latency of ML/DL algorithms execution	*Target Value> = 30%* Even if this KPI is not targeting the work carried out under this task, its outcomes however will be used as the technology pillar of the work that is currently carried out under the scope of the tasks that this KPI is targeting. This deliverable has demonstrated how the response time of the execution of an analytical operation can be improved in orders of magnitude, that will increase overall reduction in latency of the ML/DL algorithms that make use of it.
Reduction of Effort for development of ML/DL functionalities in the Sector	*Target Value >=50%* Even if this KPI is not targeting the work carried out under this task, its outcomes however will be used as the technology pillar of the work that is currently carried out under the scope of the tasks that this KPI is targeting. Having declared the *online aggregates* presented in this task, the data analyst can avoid the use of complex architectures for the development of his or her ML/DL functionalities and instead, can rely on the outcomes of this task and make use of the INFINISTORE as the single component for ingesting data at very high rates (exploiting the results of T3.1) and performing heavy analytical operations at the same time on the operational datastore.

8 References

[1]. Tamer Özsu, Patrick Valduriez. Principles of Distributed Database Systems, 4th Edition, Springer, 2020
[2]. Squirrel, SQL UI client, http://squirrel-sql.sourceforge.net/
[3]. DBeaver, SQL UI client, https://dbeaver.io/
[4]. What is a lambda architecture? https://databricks.com/glossary/lambda-architecture