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

Infinitech

D3.1 – Hybrid Transactional/Analytics Processing for Finance and Insurance Applications - I

Lead Beneficiary	**LXS**
Due Date	2020-08-31
Delivered Date	2020-08-28
Revision Number	3.0
Dissemination Level	Public (PU)
Type	Report (R)
Document Status	Release
Review Status	Internally Reviewed and Quality Assurance Reviewed
Document Acceptance	WP Leader Accepted and Coordinator Accepted
EC Project Officer	Pierre-Paul Sondag

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]	Name Surname[2]
LXS	Lead Beneficiary	Ricardo Jiménez-Peris
LXS	Contributor	Boyan Kolev, Javier López Moratalla, Patricio Martinez, Sandra Ebro, Alejandro Ramiro, Jacob Roldan José María Zaragoza, Jesús Manuel Gallego
UPRC	Contributor	Ioannis Kranas
UBI	Contributor	Konstantinos Perakis, Dimitris Miltiadou
UNP	Contributor	Bruno Almeida Tiago Teixeira
FI	Internal Reviewer	Gisela Sanchez, Lucile Aniksztejn
IBM	Internal Reviewer	Fabiana Fournier
INNOV	Quality Assurance	Dimitris Drakoulis

Revision History

Version	Date	Partner(s)	Description
0.1	2020-08-05	LXS	ToC Version
0.2	2020-08-10	LXS, UPRC, UBI	Adds introduction section and executive summary
0.3	2020-08-18	LXS, UNP, ORT	Adds section 2
0.4	2020-08-19	LXS, UPRC	Adds section 3
0.5	2020-08-20	LXS	Adds section 4
0.6	2020-08-20	LXS	Adds conclusion
0.7	2020-08-20	LXS	Finalizes the document
1.0	2020-08-20	LXS	First Version for Internal Review
1.1	2020-08-22	IBM	Internal review
1.2	2020-08-26	FI	Internal review
2.0	2020-08-27	LXS	Version for Quality Assurance
3.0	2020-08-28	INNOV	Submitted

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

[2] Can be left void

Executive Summary

The goal of task T3.1 “Framework for Seamless Data Management and HTAP” is to provide a seamless way for data management across operational and analytical data stores by supporting the Hybrid Transactional and Analytical Processing (HTAP). The importance of this task is summarized in the elimination of the need to build and maintain different types of datastores that support the two different workloads: operational and analytical. Traditional database management systems can provide support to either the operational or the analytical workload but are very underperforming when an hybrid load should be served. Due to this, data are being moved from the operational datastores that ensure transactional semantics to the analytical database management systems, which allows for read-only operations; however, they cannot support transactions, and therefore, write operations. Moving data from one store to the other is being done by a process that is called ETL (extract, transform, and load), which is cost and time consuming. Another drawback is that this is a batch process, often taking place during the night where the operational workload is very low, that ends up with having multiple copies of the dataset across the different datastores (the operational one, and the data warehouse). Moreover, the analytical queries take into account old and obsolete data from the last day, and therefore, the results of the analytical processing cannot rely on live data that have been generated and inserted in real-time.

In order to overcome the above obstacles, we will implement HTAP at the data management layer of the INFINITECH platform, in order to provide a seamless way to access data coming from both worlds: operational data with historical data that have been stored in a data warehouse. Having a single data platform that can handle both workloads is crucial for applications and analytical processing needed by the finance and insurance sector, as it is getting more important than ever the need to provide real-time business intelligence that relies on live data, and the key element to provide this is the support of hybrid data workloads on the same dataset: both operational and analytical ones.

This deliverable describes the initial steps for the INFINITECH HTAP design and implementation. At this phase of the project, the transactional behavior of the data management layer supports the concept of the *snapshot isolation* paradigm, that is the key to allow both operational and analytical processing, which are contradictive and the one competes the other while accessing data elements in the storage layer. Having done this, the HTAP can be feasible from the INFINITECH platform. Moreover, the basic architectural design of the OLAP engine of the platform has been delivered, which will allow for the effective execution of analytical queries in order to compete the performance of traditional data warehouses. The work that has been carried out in the scope of this task until M11, along with the theoretical background that justifies it are presented in this document. It is important to highlight that this is the first version of the deliverable, and two more versions (i.e. INFINITECH deliverables D3.2 and D3.3) will be published correspondingly in the next reporting periods. They are planned to include the overall implementation, along with detailed results of the benchmarking process.

Table of Contents

List of Figures

Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
BI	Business Intelligence
CflM	Conflict Manager
CgM	Configuration Manager
CmS	Commit Sequencer
CPU	Central Processing Unit
DoA	Description of Action
DL	Deep Learning
ETL	Extract, Transform, Load
HTAP	Hybrid Transactional and Analytical Processing
I/O	Input / Output
IoT	Internet of Things
JDBC	Java DataBase Connectivity
JSON	Javascript Object Notation
LTM	Local Transaction Manager
ML	Machine Learning
MVCC	Multi Version Concurrency Control
OASIS	Organization for the Advancement of Structured Information Standards
OLAP	Online Analytical Processing
OLTP	Online Transactional Processing
REST	Representational state transfer
SnS	Snapshot Server
SQL	Structured Query Language
TPC	Transaction Processing Council
WP	Work Package

1. Introduction

This deliverable summarizes the work that has been done in the scope of task T3.1 “Framework for Seamless Data Management and HTAP” at the first phase of the project (M11). The goal of this task is to provide a seamless way to access data that are usually stored in operational database management systems and data warehouses, by eliminating the need to support and maintain both types of datastores and having all data kept in a single platform. By doing this, the data will be stored once, instead of having multiple copies of the data elements being kept across the different datastores. Moreover, it will be feasible for the analytical processing to scan and take account on live data, instead of performing the analytics on a snapshot of the dataset, as it has been imported to the data warehouse from the last periodic execution of the batch ETL (extract, transform, and load) process that copies data from the operational store to the data warehouse. By doing so, real-time business intelligence, that is becoming crucial for various cases coming from the insurance and finance sector and are addressed by the INFITECH platform, can become a reality.

Operational database management systems ensure transactional semantics, which means that they support ACID properties. They enable *atomic* execution of a series of operations inside a transaction (the *A* in those properties), which means they will be either executed all or none. They ensure data *consistency* (the C in the properties)*,* which means that they will leave the database in a consistent state after a transition from one state to another, which is being done after a data modification operation. They allow for the *isolation* of transaction (the I in the ACID) which implies that the result of the concurrent execution of a number of transactions will be equivalent as if they were executed isolated, one after the other. Finally, they ensure *durability* of the dataset (the D in the ACID properties) which implies that the dataset will be durable and can be recovered in a case of failure, after the successful commitment of a given transaction. In order to support the *isolation* property, traditional database management systems often make use of a two-phase locking mechanism. This mechanism introduces shared and exclusive *locks* while accessing a data element. When an operational transaction is being executed, a data modification operation on a data element (e.g. update the current balance of a client account) is performed, which introduces an exclusive lock on that element. Exclusive locks forbid concurrent operations to access that element. Therefore, when the data analyst wants to check the overall balance of a client to recommend a new product, this operation is being blocked until the former transaction is committed successfully. As a result, analytical workloads that need to fully scan a dataset are being blocked by the operational loads that are crucial for a finance institution to ensure the consistency of the finance transactions of their clients.

As online finance transactions are being executed daily, it is very crucial to ensure data consistency and isolation when these transactions are being performed in parallel. To give an example, when the end-user makes a money transfer, it should be ensured that there is enough balance on her account to perform this task. Operational database management systems provide this type of insurance, often by introducing some type of locking mechanism on the data elements that are being updated, with the cost of down perform in cases of analytical processing. The latter often requires the full scan of a given dataset, which is blocked by the various locks imposed by the operational processing. In order to overcome this, the data administrators perform periodically some type of ETLs that migrate data from the operational datastores to the data warehouses. The latter support read only operations, and therefore, allow for a full scan of a dataset, without being blocked by concurrent data modification operations. However, they rely on a snapshot of the dataset of the previous day, or from the last execution of this periodic batch migration, which can be inadequate in cases that there is the need to produce results in real-time. This is the case for the online identification of a fraud behavior. The analysis needs to take place on the live data, as they are inserted into the system to make the identification useful for performing an immediate action. Another example will be the risk assessment of a possible recommendation that may need to take into account data from the current day, and not to rely on a previous snapshot that might give erroneous cost estimation. An analytical processing from an insurance organization might also need to take into account sensor IoT data as they are inserted into the system, rather than wait for the data to be injected into a data warehouse for further processing.

As has been already mentioned, in order to overcome this problem, data is being migrated periodically to a data warehouse, which supports analytical processing, as it does not lock data elements. Data are being migrated to the warehouse usually during the night, where insurance institutions usually experience very low operational traffic, and therefore, it is feasible to block all those operations for a certain period of times while moving data from the one store to the warehouse. This block/downtime would be inacceptable to be executed by day, as it would block the whole organization. Apart from copying data into multiple destinations, the additional obvious result of this migration process, often called ETL, is that it has to be executed periodically during night, thus, the data analyst can rely only on a snapshot of the dataset of the previous day. The drawback of this approach is that it can only support *near* real-time business intelligence (BI), which is mostly accepted in the majority of the use cases. However, the scope of the INFINITECH project is to provide real-time BI and therefore task T3.1 supports Hybrid Transactional and Analytical Processing (HTAP) that is crucial for the needs of the modern enterprises from the insurance and finance sectors.

1.1. Objective of the Deliverable

The objective of this deliverable is to report the work that has been done in the context of the task T3.1 at this phase of the project (M11). This task lasts until M27, and therefore, two more versions will be released, extending and modifying, when necessary, the content of this document, following the agile approach for system development and in order to update the solution and implementation with the current trends of the environment as the project progresses. The work that has been done during this phase (M03-M11) was mainly focused on the **delivery of the core transactional component of the data management layer of the INFINITECH platform**, which enables the seamless data access over the HTAP workloads. Instead of using traditional two-phase locking implementations to ensure the ACID properties, our implementation makes use of the *snapshot isolation* paradigm that avoids locking and therefore allows for i) the scalability of the transactions that can now support hundreds of millions of concurrent operations, and for ii) the concurrent execution of read-only operations over the same dataset, which are used by the analytical tools. In addition, an overview of the OLAP (Online Analytical Processing) engine of the data management layer of the platform is being presented.

According to the project workplan, in the forthcoming period, the HTAP engine of the INFINITECH platform will be validated against the various use cases that need this capability. An extensive benchmarking will also take place combining our solution with native operational and analytical database management systems, using the family of the TPC-* benchmarks[3]. Those benchmarks have been defined by a group of dominant database vendors and provide a series of transactions that can be usually found in the majority of enterprise applications. They define a common way to benchmark database management systems. The plan is to make use of the TPC-C benchmark that is ideal for operational workloads, and the TPC-H that has been designed to stress and benchmark analytical workloads. We will also make use of the mixed TPC-CH which provides a combination of

[3] Transaction Processing Council Benchmarks: http://www.tpc.org/tpc_app/

those two: In particular, the TPC-H has been modified in order to match the schema of the TPC-C, and provides the same sets of operations as the original one. Having those three available, we plan to execute the TPC-CH over our HTAP implementation and compare the analytical results with the execution of the TPC-H over an OLAP engine and the TPC-C over an operational datastore. This work however is planned to be delivered on the forthcoming versions of this deliverable.

1.2. Insights from other Tasks and Deliverables

The work that is reported in this deliverable is based on the overview description of the corresponding task T3.1, which has been further specified in more details at the WP2 level, which is the fundamental work package that defines the overall requirements for the whole platform. In more detail, task T2.3 comes with the specification of the technologies that the overall platform of INFINITECH provides aiming to specify the technical requirements that need to be covered by the technical tasks of the WP3-4-5-6 work packages. Task T2.5 additionally provides the definition of the various datasets that the overall data management layer must support and must be taking into account by this task that implements the core engine of this layer. Moreover, the Reference Architecture (RA) of INFINITECH is being defined in T2.7 and the work that is being done in the scope of this task must be fitted into the whole design. Therefore, as a technical task, T3.1 has clear dependencies with T2.7. Finally, T3.1 takes input of the whole WP7 where the definition of all use cases takes place. On the other hand, as T3.1 implements the fundamental core of the overall data management component, it is related with the majority of the other technical tasks. More precisely, it gives output to T3.2 which is related to the polyglot extensions that access data on real-time through this layer. Tasks T3.3 and T3.4 need to correlate streaming data with the data *at-rest*, therefore the HTAP capabilities are crucial when having to perform real-time streaming processing and combine these two different types of data. Moreover, T5.1 collects data and ingests them into the platform, therefore the scalability of the transactional processing of T3.1 comes in place, while the remaining tasks T5.2, T5.3 and T5.4 are related to tools for analytical processing that need to take into account real data, where HTAP capabilities allows for that.

1.3. Structure

This document is structured as follows: Section 1 introduces the document, putting the work reported in this deliverable under the context of the project, highlighting its relation with other tasks of the DoA. Section **Error! Reference source not found.** provides the fundamental theory and explanation on how we make feasible the Hybrid Transactional and Analytical Processing. Section **Error! Reference source not found.** describes how the transactional processing is being implemented in the scope of the INFINITECH platform, that allows for both the scalability of the data modification operations while ensuring the transactional semantics, which is crucial when we need to deal with real-time data, instead of putting them into a queue and periodically batch import them to the datastore, while section **Error! Reference source not found.** presents the overall design of the OLAP engine. Section **Error! Reference source not found.** concludes the document.

2. Making Hybrid Transactional and Analytical Processing Feasible

A crucial requirement for the majority of the applications and processing coming from the insurance and finance sector is to ensure transactional processing, meaning that the operational workload must ensure the ACID (Atomicity, Consistency, Isolation and Durability) properties. The concept of a database transaction ensures those properties for the lifecycle of the transaction. Transactions are a very important abstraction in developing applications since they remove the complexity of difficult concepts from the application layer, down to the database. When dealing with concurrency, users do not need to take care about the concurrency control needed when developing applications and software components that need to bracket the access to shared data. In fact, it is the corresponding implementation of the protocols in the database level that ensures the corresponding isolation property and takes care about the details of concurrent access. Secondly, software developers do not have to deal with failures, as *atomicity* and *durability* protocols provide automated recovery in the advent of failures yielding all-or-nothing semantics. As a result, the ACID properties simplify the task of programmers.

In this chapter, we focus on the *isolation* property and how different levels of isolation of concurrent transactions affect the results of a query when being executed in parallel. The different implementations of the management of concurrent transactions in order to implement the defined isolation level, affects the ability of the data management layer to provide hybrid transactional and analytical processing. We give an overview of the different isolation levels first, in order for the viewer to deeply understand the problem, along with the corresponding read phenomena that are subject to each level. Then, we describe how these are achieved by the two dominant approaches that are implemented by traditional database management systems: the *two-phase locking* approach and the *snapshot isolation* paradigms.

2.1 Isolation Levels and Read Phenomena

In database theory, the term *isolation* defines the visibility of data elements to concurrent transactions and if a user transaction can modify an element that has been previously accessed by a concurrent one, or has the visibility of a data modification that another concurrent transaction has performed. To give an example, a user wants to perform a money transfer from one of her account to another, while on the same time she tries to buy a market product that will imply the invocation of a finance transaction from one of her accounts to buy this new product. Those two transactions try to access the same data element (the value of her account that has been written and persistently stored in a database management system), however it is up to the level of *isolation* to decide whether or not both of them can perform the operation.

Typically, lower isolation levels increase the ability of many concurrent transactions to access the shared data. However, the lower the level of isolation, the more important the read phenomena that are allowed to happen. In our previous example, a money transfer from one of the user's account to the other might leave her primary bank account with no money. While executing the finance transaction to buy a new product on the same time, if the isolation level is low enough, it might be possible that this transaction will have the visibility of the old value of the bank account, and therefore, perform the transaction to buy the product, while in reality, the amount of the required money has been already moved to the new account. These types of concurrent operations are most likely found in typical applications of the finance and insurance sectors and they require a higher level of *isolation.* As it has been already stated, the higher the level *isolation* is, the lower the level of concurrent access. To make things worse, in many cases, it might be impossible for an analytical operation to be executed when the level of *isolation* is very high, which is a requirement for finance institutions. This explains the need to migrate data periodically to data warehouses that only allows for read only operations where transactional semantics are not imposed.

Typical isolation levels are the following:

- Read Uncommitted: An on-going transaction can read the modified value of a data element that has been recently modified by a concurrent transaction that has not yet been committed. This is the lower level of *isolation* which provides no insurances that the value that the first transaction has read will be valid at the end of its lifecycle. It can be used in cases where the validation of the data is not a priority, rather than the level of concurrency.
- Read Committed: This is the default isolation level supported by the majority of operational datastores. It ensures a minimum consistency of the data that are being concurrently accessed and covers the majority of the use cases. It allows for a transaction to read the value of a data element, when a concurrent transaction that has previously modified its value is now committed. It usually forces the read operation to be blocked until the concurrent one is successfully committed. However, it allows for some read phenomena that are inacceptable for the finance sector.
- Repeatable Reads: In this isolation level, the concurrency control mechanism ensures the validation of the value that has been once read during the whole lifecycle of the transaction. It forbids any concurrent transaction to modify the value of data element that has been previously read by an on-going transaction. This will usually require the lock of that data element that reduces the level of concurrency and as a result, downgrades the overall performance.
- Serializable: This is the highest isolation level and ensures that concurrent transactions are being executed as they were occurred in order, rather than in parallel. In most applications of the insurance and finance sector, this is the required level of isolation that must be ensured by the data management level. However, this implies that instead of a concurrent execution of transactions, they will be executed sequentially.

As it has been noted, each one of the aforementioned isolation levels allows for different read phenomena. Let's examine them per level.

Dirty Reads

These phenomena can occur under the *read uncommitted* isolation level. The following diagram highlights the phenomenon.

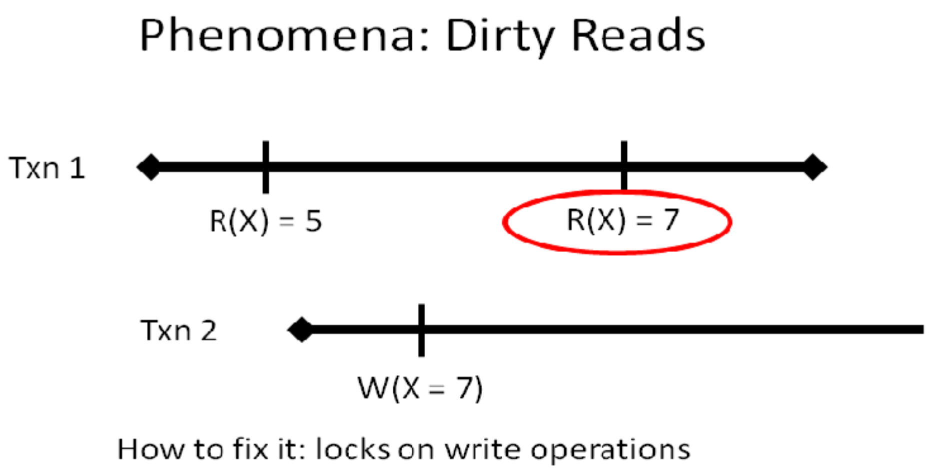
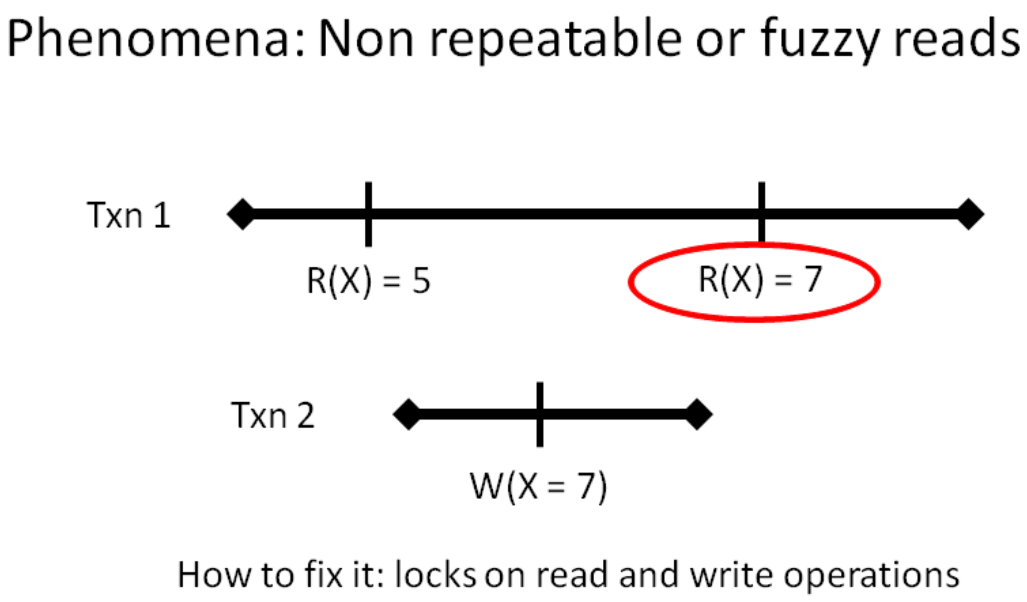


Figure 1: Dirty Reads phenomenon

An on-going transaction T1 reads a data element X while a concurrent transaction T2 writes the value on the same data element X. T1 now reads again the value of X that has been modified to its new value 7. However, T2 aborts and T1 has read a dirty value 7 for the data element X. This can have a crucial effect in a finance organization when T2 moves money from one account to another, and T1 performs a finance transaction in order to buy a product. It assumes that the bank account of the user has value 7 which might be enough to perform the finance transaction, but this is not true, as T2 fails and the user's account has in fact value 5, which is not adequate enough to buy the product. As we will see in the next subsection, this can be solved by putting exclusive write locks on the modified data elements.

Non Repeatable Reads

These phenomena can occur under the *read committed* isolation level. The following diagram highlights the phenomenon.



Phenomena: Non repeatable or fuzzy reads

How to fix it: locks on read and write operations

Figure 2: Non Repeatable Reads phenomenon

An on-going transaction T1 reads a data element X while a concurrent transaction T2 writes the value on the same data element X. T1 now tries to read again the value of X that has been modified to its new value 7. However, under this isolation level, it has to wait until T2 firstly successfully commits. Then it is ensured that the value that T2 will read is valid, and therefore the check whether the user has enough money on her bank account to perform the finance transaction. Even if this improves the phenomenon that was noticed with the *read uncommitted* isolation level, it still has a severe implication when it comes to operations related to the finance and insurance sector. T1 initially read the value of the data element that was 5, and when it tried to read it again, it has been changed to 7. As a result, it does not allow for repeatable reads in the same transaction. Taken into account that the operations inside a transaction must be atomic, which means they either need to be executed all or none, they also need to have the same visibility on the same data items. This might have severe implications when, for instance, the finance institution starts a transaction that firstly reads the value of the account of the user to decide whether or not she is allowed to perform the finance operation. It might need to do this at the beginning in order to avoid cost-demanding write operations afterwards. However, in the meantime, T2 has modified the value of its account, and when it actually performs the operation, it reads a different value that is not consistent in the scope of T1. As we will see in the next subsection, this can be solved by putting exclusive write locks on the modified data elements, along with shared read locks on the elements that have been accessed by a read operation.

Phantom Reads

These phenomena can occur under the *repeatable read* isolation level. The following diagram highlights the phenomenon.

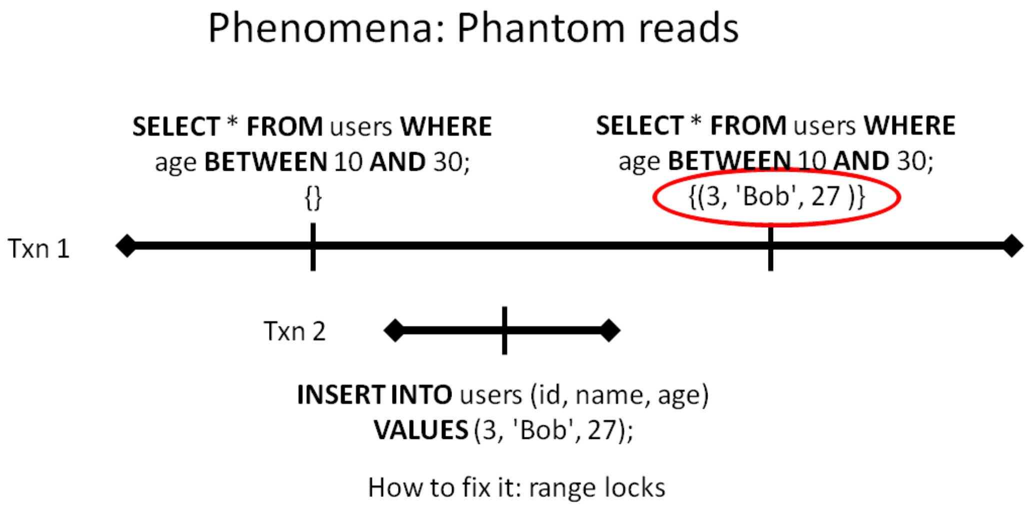

Phenomena: Phantom reads
SELECT * FROM users WHERE age BETWEEN 10 AND 30;
{}
SELECT * FROM users WHERE age BETWEEN 10 AND 30;
{(3, 'Bob', 27)}
Txn 1
Txn 2
INSERT INTO users (id, name, age) VALUES (3, 'Bob', 27);
How to fix it: range locks

Figure 3: Phantom reads phenomenon

These phenomena occur when we have transactions that require scan operations on a dataset. In our example, an on-going transaction performs a scan operation to return the data elements whose field *age* is between the values 10 and 30. A concurrent transaction however inserts a new data element whose *age* value is 27 and commits. When T1 will execute the same query, it will return a different result set, as a *phantom* record has been added in the meantime. This might be important in finance operations that calculate for instance the overall spend of a client, by calculating the cost value of all her transactions during the last week. That might be meaningful for a fraud detection mechanism. However, if in the meantime, the user performs a massive money transfer, the result of two sequentially executions of the statement inside the lifecycle of T1 will be invalid. As we will see in the next subsection, this can be solved by putting range locks that will forbid a concurrent transaction to insert a new value inside this range. However, this can be very ineffective and might typically require the construction of a new index on a specific field and possibly, the lock of the entire table during the execution of the scan operation, that will have a side effect to entire block of all write operations that need to access this dataset. In order to overcome this, the data administrator migrates data on a data warehouse and performs these types of read operations there, with the drawbacks that have been mentioned in the previous sections.

2.2 Two-Phase Locking

Traditional operational database management systems make an extensive use of the *two-phase locking* mechanism in order to ensure the different isolation levels. It is a concurrency control mechanism that makes use of different types of locks on data elements, thus blocking an on-going transaction from accessing a shared data element that a concurrent one has previously accessed. The details of the implementation of this mechanism are out of the scope of this document, but it is important to mention that it involves two phases: the first one where the locks are acquired while accessing the data, and the second one where the locks are released while the transaction is committed. There are several variations on the protocol, but all share the same concept: the introduction of locks on data elements that can be of two types:

- Shared or read locks: this is added when a transaction is trying to access a data element to perform a *read* operation.

- Exclusive or write locks: this is added when a transaction is trying to access a data element to perform a *data modification* operation.

An exclusive lock prevents all other operations to access the specific data element until the lock is released. This means that when modifying a data element, no other read or write operation can be performed. This prevents from write-write and write-conflicts to happen. On the other side, a shared lock only forbids a write operation to access a data element, thus preventing a read-write conflict. However, other read operations are permitted to access the data element and in fact, read-read operations are allowed as there cannot be conflicts.

In the scope of the different levels of isolation, these locks can help achieve the desired level. *Dirty reads* phenomena can be avoided by introducing an exclusive lock on the data element that has been modified. By doing so, a concurrent read operation T2 is not allowed to access the previously modified data, as the exclusive lock blocks its access. It has to wait to either the on-going transaction commits or aborts, which will have the consequence of releasing the corresponding lock. At that time, the concurrent transaction T2 will be allowed to access the data element, and according to Figure 1 it will read either the value 7, if T1 has committed, or the value 5, if T1 has been aborted. In any case, the exclusive lock prevents the dirty reads phenomena and allows for a concurrent transaction to always read a valid value.

In cases of *non-repeatable reads* phenomena, they can be prevented by the use of shared locks. According to Figure 2, the on-going transaction T1 accesses a data element X and locks it with a shared lock. This will prevent the T2 to modify its value and as a result, T1 will later read the same value 5 as previously. When T2 commits, it releases its locks and T2 can now modify the value of the data element X. It is important to notice that in the case of a concurrent transaction T3 that wants to read data element X, this will be feasible and will read the value 5, as T2 is still blocked by the lock added by T1, and shared locks does not block read operations.

In cases of *phantom reads* phenomena, a similar mechanism is also applied. However, according to Figure 3, a list of shared locks is being applied in the range that affects the read operation. It is true that if a shared lock is being applied in each of the accessed values, this will still permit a concurrent T2 transaction to insert a new data element, as this element will not have been previously locked by T1. In that case, T1 will receive a different result set with a *phantom* element added by T2. In order to prevent this, a scan operation adds shared locks in a range of values, preventing all concurrent operations to modify and insert elements in this range. According to the type of field that needs to be scanned and the implementation of the corresponding database management system (e.g. locking on the data element level, locking on the leaf of the corresponding index, locking on the data table level, etc.) this can block the entire operational workloads that needs to access of a specific table.

Even if the corresponding isolation level can be achieved by the use of shared and exclusive locks introduced by the *two-phase locking* concurrency control, it also introduces two significant inherit obstacles: the maintenance of the locks requires a central component that orchestrates the whole process and coordinates the distribution of locks across different data shards in a distributed deployment. As this component is central, it cannot scale adequately and becomes a bottleneck when there is the need to scale out the database to multiple nodes. In fact, most of the traditional database management systems can scale out to a certain degree as the improvement of the performance of the overall system reaches its peak and starts to be downgraded. Moreover, the use of range locks for scan operations that are required by the analytical processing further blocks all write operations on this datasets, making the whole application to down perform or to completely block in case a data analyst tries to make an analysis on the live data. As it has been noted, shared and exclusive locks are contradictory and in fact, analytical workloads adding shared locks compete with the operational workloads adding exclusive locks. This prevents HTAP to happen and therefore, the data administrators make use of expensive ETLs to move the operational data periodically to a data warehouse that allows read only operations without locking.

As the INFINITECH data management component needs to provide HTAP capabilities, it will rely on a novel and recently used paradigm that follows a different approach, which is called *snapshot isolation* and allows for the concurrent existence of both loads, as it removes the need for locking and therefore, it never blocks transactions.

2.3 Snapshot Isolation

Snapshot isolation[4] exists since a long time ago; however it became popular during the last decade where the need for scalability was raised up due to the wider adoption of cloud applications. This, combined with the need to continue to ensure transactional semantics made the traditional two phase locking mechanism inappropriate for distributed database management systems. *Snapshot Isolation* provides a very high isolation level due to the fact that transactions read from a snapshot of the database with the state, as it was when the transaction was started. To this end, this paradigm requires the use of multi-version concurrency control. By using this mechanism, instead of storing a single version of each data item, a new version is created when a transaction that updated the item commits. Therefore, for a single data item multiple versions of it can exist at a given time. These versions need to be labelled in a way that they enable to choose the right version for a given transaction that tries to read a data item. Typically, logical timestamps are used for this labelling.

In order to implement the *snapshot isolation*, a unique component is responsible to distribute these logical timestamps. Transactions are assigned with those timestamps both when they start and when they commit. In order to understand how this protocol works, let’s look at Figure 4.

[4] EuroSys '12: Proceedings of the 7th ACM european conference on Computer Systems, April 2012 Pages 155–168, https://doi.org/10.1145/2168836.2168853

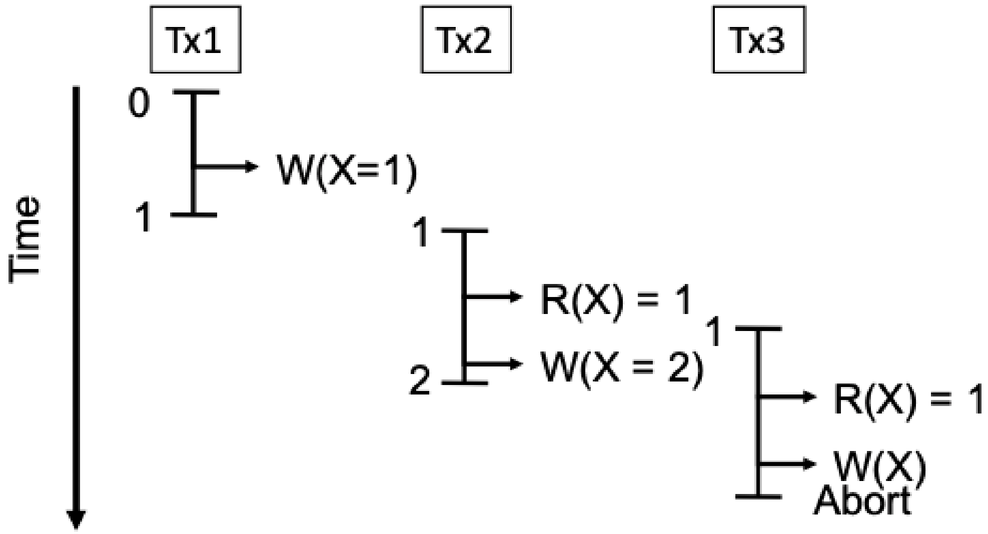


Figure 4: Snapshot Isolation in practice

Let’s assume that we have a central component responsible for distributing the logical timestamps. This component gives the current value of the timestamp when a transaction starts and increases this value when a transaction commits. Let’s assume that there is a data element X at snapshot 0 with value 5. Transaction T1 starts at the timestamp 0 and gets assigned with that value. Then it performs a write operation on the data element and modifies its value to 1. Assuming there are not write-write conflicts, it commits. At commit time, the *timestamp distributor* increases the value to 1, and assigns this value to T1. T1 marks the data element with this timestamp and stores its value. Now data element X has the old version at timestamp 0 with value 5 and a new version at timestamp 1 whose value is 1.

T2 now starts. It is assigned with the current value of the timestamp which has been now forwarded to 1. It tries to read data element X. It will read the most recent version/snapshot of this data element according to its *start timestamp.* Data element X has two versions: 0 and 1. The most recent to timestamp 1 is the version 1, and therefore, it reads the corresponding value which is 1. It then modifies its value to 2 and it tries to commit. As there is no write-write conflict at that time, the distributor forwards the timestamp to 2, and assigns that value at commit time to T2, which in turns, creates a new version/snapshot of that value, whose value at timestamp 2 will be 2. Therefore, now data element has three versions: version0 with value 5, version1 with value1 and version2 with value 2.

However, a concurrent transaction T3 has been already started before T2 commits. It will be assigned with that *start timestamp* of the current value, which at that point, was 1. Even if it is concurrent with T2, it will try to read data element X after T2 commits. As it has been assigned the timestamp 1, it will check for the latest version of the data element X before the timestamp it has. Data element X has now 3 versions, and the latest one to timestamp 1 is the one whose timestamp 1 one. Therefore, it will read the value 1 that corresponds to version 1 and not the value 2 which corresponds to version 2, which has been created after T3 started. As a result, there is no need to check for read-write or write-read conflicts. Later on, it will try to modify the value of that data element. Here, a write-write conflict will be identified, as the latest version of data element X now is 2, which later than the *start timestamp* of T3 which is 1. T3 will have to abort.

It is important to mention that even if snapshot isolation avoids all read-write conflicts including the aforementioned one between predicate reads and writes, it still forbids write-write conflicts. This requires for checking those conflicts with some conflict management system.

We can see from Figure 4 that the *snapshot isolation* paradigm makes no use of locking and therefore permits read operations to scan a dataset on the same time with operational load taking place concurrently on the same dataset. *Dirty* and *Non-Repeatable reads* phenomena cannot happen as there is no need for read-write or write-read conflicts; the protocol itself ensures that each transaction will read the corresponding version of the data element. *Phantom reads* phenomena are also removed, as a repeatable scan operation will never see a *phantom* element added by a concurrent transaction, as the latter will always have a timestamp (and therefore a version) bigger than the on-going scan operation. It comes with the drawback of handling multiple versions of data items, however a *garbage collector* can be used that removes old and not accessible versions. The INFINITECH data management layer makes use of this paradigm in order to allow the Hybrid Transactional and Analytical Processing (HTAP) to be feasible.

3. Transactional Processing in INFINITECH

The INFINITECH data management layer is relied on the *snapshot isolation* paradigm in order to efficiently handle transactional processing. This gives us two benefits: firstly, by design, it allows for Hybrid Transactional and Operational Processing (HTAP) and secondly, it removes the inherit bottleneck that is introduced by traditional *two-phase locking* mechanisms due to the maintenance of the locks across the distributed nodes. However, traditional implementations of the *snapshot* isolation tend not to scale out effectively, and thus, cannot support operational (OLTP) workloads in very high rates. With our design, the data management layer is capable to scale out to hundreds of data nodes that allows it to server operational workload in very high rate. The following subsections provide more details regarding the implementation of our approach.

3.1 Centralized Transactional processing

A transaction can be seen as a sequence of read and write operations on data elements. It must ensure ACID properties, which means the transactional database management systems must provide atomicity, consistency, durability, and isolation. As a result, when a transaction commits, all the data modifications are guaranteed to be durable. If this is not possible due to a commit failure or a write-write conflict in one of the involved operations, the transaction must abort and none of its updates should become visible to other transactions. As it was mentioned in the previous section, the concurrency control mechanism that orchestrates the execution of concurrent transactions relies on the snapshot isolation.

In order to implement the snapshot isolation paradigm, we need to implement the corresponding Multi-Version Concurrency Control (MVCC). In our implementation each write operation wi(xi) of transaction Ti on record x creates a new private version x, and each read operation ri(xj) of transaction Ti reads the latest version of x, xj created by a committed transaction Tj such that j<i and there is no other committed transaction Tz, such that j< z<i. Snapshot read requires that a transaction Ti reads a snapshot of the database that reflects the latest committed versions of all records as of start time of Ti. In particular, this means that if Ti performs a read ri on x, then it reads either the private version Ti previously created (*read your own writes*) or it reads the version xj created by Tj such that Tj was the last transaction to write x and commit before Ti started. Snapshot write requires that no two concurrent transactions (i.e., neither committed before the other started) update the same entity. If this happens, one of the two transactions will abort (typical strategies are either the *first-committer-wins*, or the *first-updater-wins*). As it was highlighted in the previous subsection, in case a transaction commits, then the current commit timestamp is increased and the private versions that correspond to the data modifications that the transaction has done are marked with this timestamp and then are sent to the datastore to be persistently stored. If the transaction fails and aborts for whatever reason, this private write-set is released.

Figure 5 illustrates the interaction inside the data management layer of INFINITECH of its various components. There are depicted three major ones: the data storage layer, the query processing layer that requests data from the data storage, and the transactional management layer, that ensures the ACID properties and provides the transactional semantics needed by the operational workloads. In this simplified version of the implementation which is compatible with the logical processing of the data management platform itself, let's assume that the query processing layer reads data from the data storage, performs the updates locally and creates the private write-set, and only writes data back to the data storage upon commit. Variations of this protocol can be also found, where the private write-set can be written in the data store layer but it is visible only to the corresponding transaction that has written it, unless the commit takes place where the visibility is guaranteed to all transactions.

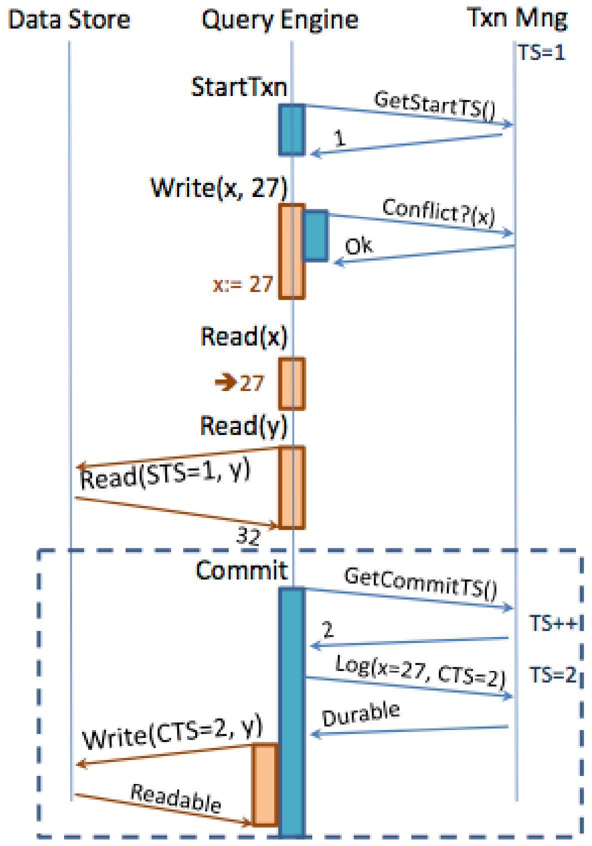


Figure 5: Centralized Transaction Processing

Let's assume that the current commit timestamp maintained by the transactional manager is TS=1.

When a transaction T_i begins, it asks for this value. In case T_i wants to perform a write operation, it has to ask the transaction manager to check if there are write-write conflicts with the corresponding data element. A conflict can occur if there is a concurrent transaction Tj, i.e., Tj has not committed yet or its commit timestamp is larger than Ti's start timestamp (C(Tj) > S(Ti)), and Tj has written x. If there is no conflict, a private version of x is being maintained by the on-going transaction and will be visible only to this, until commit time. If there is a conflict, according to the *first-updater-wins* approach, the on-going transaction must abort and release the private write-set. When a transaction Ti requests to read a record x, the data store has to provide the record created by transaction Tj with commit timestamp C(Tj), such that C(Tj) ≤ S(Ti), and there is no version of x created by a transaction Tk such that C(Tj) < C(Tk) < S(Ti). This provides the snapshot read property.

Upon commit time, the transaction asks for the *commit timestamp* and the transactional manager has to increase the value of the current timestamp. In our example, this is being increased to 2. Moreover, the updates need to become *durable*, before making the private write-set visible to other transactions. The transaction will send the log of the updates to the transactional manager, and latter persists this redo-log to a persistent storage. Only then the transaction makes its private write-set public by persistently store it to the data storage so that it can be accessible by other transactions. It is important to notice that the commit phase must be atomic, therefore the increment of the *commit timestamp* and the write of the private write-set to the datastore are tightly related; when updating the timestamp, new transactions must be able to have a visibility to the modified records. This introduces a first bottleneck while having this approach as new transactions need to wait until the private write-set is persistently stored to the data store component. As this is time consuming, the concurrent execution of transactions will be downgraded to sequential, as each transaction will have to wait for others to commit first.

Another drawback is that the transactional manager is a monolithic node performing different tasks, which might become a bottleneck in cases there is a huge number of transactions that need to be served. The transactional manager is a centralized component and cannot scale out with the current approach. Having this central component perform all these different tasks might require a lot of resources and as a result, will saturate the resources of the node where it is deployed. We need to distribute the different tasks of this component in order to be in position to scale them out independently and leave only the trivial ones that cannot be decentralized in a common node, but they will require minimum resources

3.2 Decentralized Transactional processing in INFINITECH

As it was clearly noted in the previous subsection, having a centralized component to handle the transactions gives us no benefits from the traditional database management systems that make use of the *two-phase locking* protocol. Both systems share the same bottleneck; scaling out the transactions. Even if the *snapshot isolation* gives us the ability to perform both OLAP operations on top of OLTP workloads sharing the same dataset, if the system cannot scale adequately, it will be become eventually evident the need to scale out and the proposed implementation will provide no benefits. In order to overcome this, INFINITECH provides a decentralized transactional processing mechanism that allows for:

- Scaling out to adapt to increased and diverse workloads
- Being transparent to the application developer and data analyst
- Provide adequate throughput scalability by allowing scaling out linearly
- Minimizes the latency imposed by the commit of a transaction in order to support OLTP workloads and
- Supporting the independent scaling of the other components of the data management layer: the data nodes and the query engine instances.

The following subsections give more details on how the INFINITECH data management layer supports this, making it feasible to provide HTAP capabilities.

3.2.1 Decoupling Update Visibility and Atomic Commit

As stated before, an important limitation imposed by the traditional approach is the atomic commit phase, meaning that the commit timestamp has to be increased only when the private write-set is persistently stored in the data node, which is a costly operation and downgrades the level of concurrency. In the INFINITECH data management platform, we adopt a radically different approach, by holding two different types of timestamps: the commit timestamp and the snapshot timestamp. The former is used as usual: assigns the value to transactions that commit. The later gives the value to the transaction, when it starts, called *starting timestamp.*

As a result, the sequence of operations when a transaction Ti tries to commit is now the following: Ti receives the *commit timestamp* that is incremented, then it writes its private write-set to a redo log that is flushed to the persistent storage, in order to make the transaction durable, then it writes the new versions of the data to the data nodes, in order for them to become readable from forthcoming transactions, and then it informs the transactional manager to update the *starting timestamp.* The important thing to notice in this sequence of actions is that only the first must be atomic, which is trivial, as it only involves the increment of a counter. After this action, the logging can be postponed to be executed later, and it will only affect the current Ti transaction's latency. By postponing the logging phase, it allows us to combine the logging of other transactions into one step, thus taking advantage of high throughput. After the logging phase, the transaction is now durable and in fact, we can return the control to the user, unblocking the execution of the commit and perform the next 2 steps in a later phase. At that point, whatever happens, the system can recover as the *durability* property is ensured after the successful logging. This also means that the time we need to store the private snapshots of data to the data nodes does not affect the overall latency of the transaction, and this can allow the system to take advantage of other transactions and send this information all-in-once to the persistent storage. Only when the private snapshots are stored in the data nodes, the *snapshot* or *start timestamp* can be incremented, given their visibility to forthcoming transactions. However, this delay does not affect the data consistency and is aligned with the transactional semantics.

Given that the commit phase is not executed atomically though, there might be the case where two transactions are concurrent and try to commit. T1 commits first, that is, takes a *commit timestamp* that is smaller than the one that T2 receives, however T2 private snapshots are stored earlier than T1's ones and therefore, the snapshot server is informed to advance the corresponding timestamp to the value of T2. However, this means that T1 modifications must be visible by forthcoming transactions, which is not true as T1 has not stored its private versions of data yet. In order to overcome this, when the snapshot server receives a notification to increment the corresponding timestamp, it waits until all open transactions that have a commit timestamp previous of the current one are informing its successful store of private versions, and then it increments the snapshot timestamp. Forthcoming transactions will always take the start timestamp of the enables them to have the visibility of data without losing consistency.

The snapshot counter is not incremented by one each time a transaction completes its commit. It represents the longest coherent (i.e. gap-free) prefix of committed transactions. That is, if the snapshot counter is equal to the commit timestamp C (Tj) of transaction Tj this means that data versions created by Tj and all transactions with commit timestamp smaller than C (Tj) are durable, and readable from the data store (stored in the data store layer, but not necessarily persisted).

Figure 6 shows the various phases of a transaction. When it starts, it receives the *start timestamp* from the snapshot server that provides its visibility over the data elements. That is, the transaction becomes active. When it is ready to commit, it sends a request to the transaction manager and the transaction itself becomes completed. When the logging phase is passed, then the transaction can be considered durable. After that, it sends the private versions to the data node, so that the transaction can become readable. After the notification to the snapshot server that the transaction is readable, the latter eventually increments the snapshot timestamp so that the transaction is now visible.

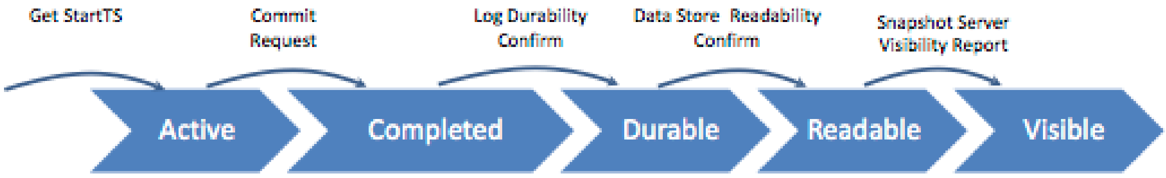


Figure 6: Transaction phases

In summary, the proposed solution that has been adopted by the INFINITECH data management layer is to process the commit phase of a transaction as a pipeline of independent tasks. That way, we can parallelize the commit processing that allows the overall system to scale in very high levels, instead of processing each commit atomically that would prevent us from parallelize it and would impose us to process each commit sequentially. The consistency of data is ensured by the snapshot server that increments the corresponding timestamp only when there is no gap between transactions that are committing but they haven't become readable yet.

3.2.2 Parallelization and Distribution

Figure 7 depicts the major architectural components of the INFINITECH data management layer. We can see that it consists of instances of the query engine, which incorporates a *local transactional manager*, the KiVi key-value internal datastore, which implements the MVCC that is a pre-requirement for the snapshot isolation to be used, and which is the persistent storage engine of the platform. Those two components can scale out independently and it is recommended that they co-exist in a node. That means, an instance of a query engine can be responsible for a couple of KiVi instances and all of them can be deployed in a single node. When it comes to scaling, we can create an additional identical node and let the storage engine redistribute the data load among its data nodes.

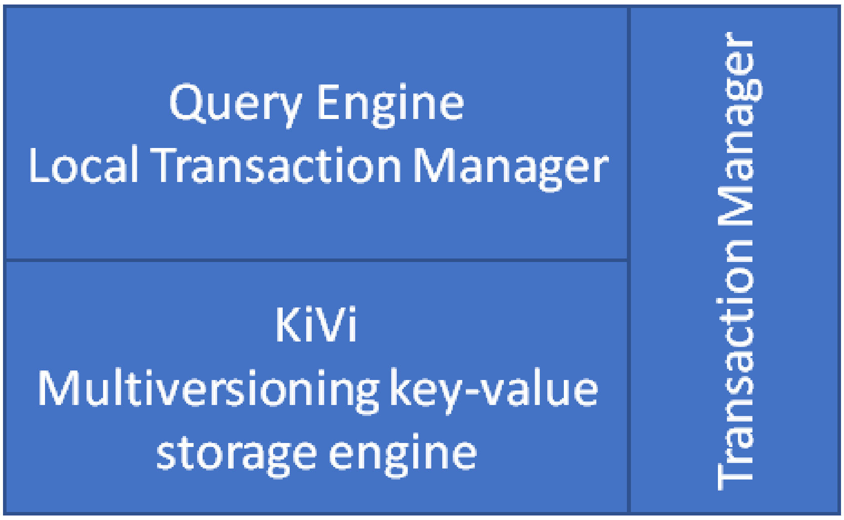

Query Engine
Local Transaction Manager
KiVi
Multiversioning key-value storage engine
Transaction Manager

Figure 7: INFINITECH Data Management Components

In addition, the data management layer also consists of the *transactional manager* that is placed vertically in the figure, meaning that is totally independent of the number of query engine or data nodes instances the overall deployment has. However, as aforementioned, having the transactional manager as a monolithic application introduces important bottlenecks as it cannot scale adequately, and thus, we have split it to independent components, each one of those is responsible for executing a specific task. The idea of distribution is to assign the relatively independent tasks executed by the transaction manager to several independent components. As a result, the transaction manager consists of the following subcomponents (see Figure 8):

- **Apache ZooKeeper (ZK)**: is responsible for coordinating the distributed process of the transactional manager. It is mainly used to send *heartbeats* to other components to identify their state and if there is a potential failure.
- **Configuration Manager (CgM)**: It holds information about the overall configuration of the system
- **Snapshot Server (SnS)**: It is informed by a transaction when the latter is readable, so that it can forward the *snapshot timestamp* accordingly in order to give the corresponding visibility to forthcoming transactions
- **Commit Sequencer (CmS)**: It is responsible for the only atomic operation that the transactional manager has, incrementing the *commit timestamp.* As a result, this component cannot scale out, but its amount of work that needs to do is tiny and cannot become a bottleneck.
- **Conflict Manager (CflM)**: It checks for write-write conflicts. More information about this component will follow.
- **Logger (LgCmS & LgLTM)**: It is responsible for storing the logging persistently to storage so that a transaction can become durable and recover in case of a failure.
- **Local Transaction Manager (LTM)**: This component is included in each instance of the query engine. It is responsible for orchestrating each transaction's state at a local level.

The deployment diagram of the distribution is depicted in Figure 8.

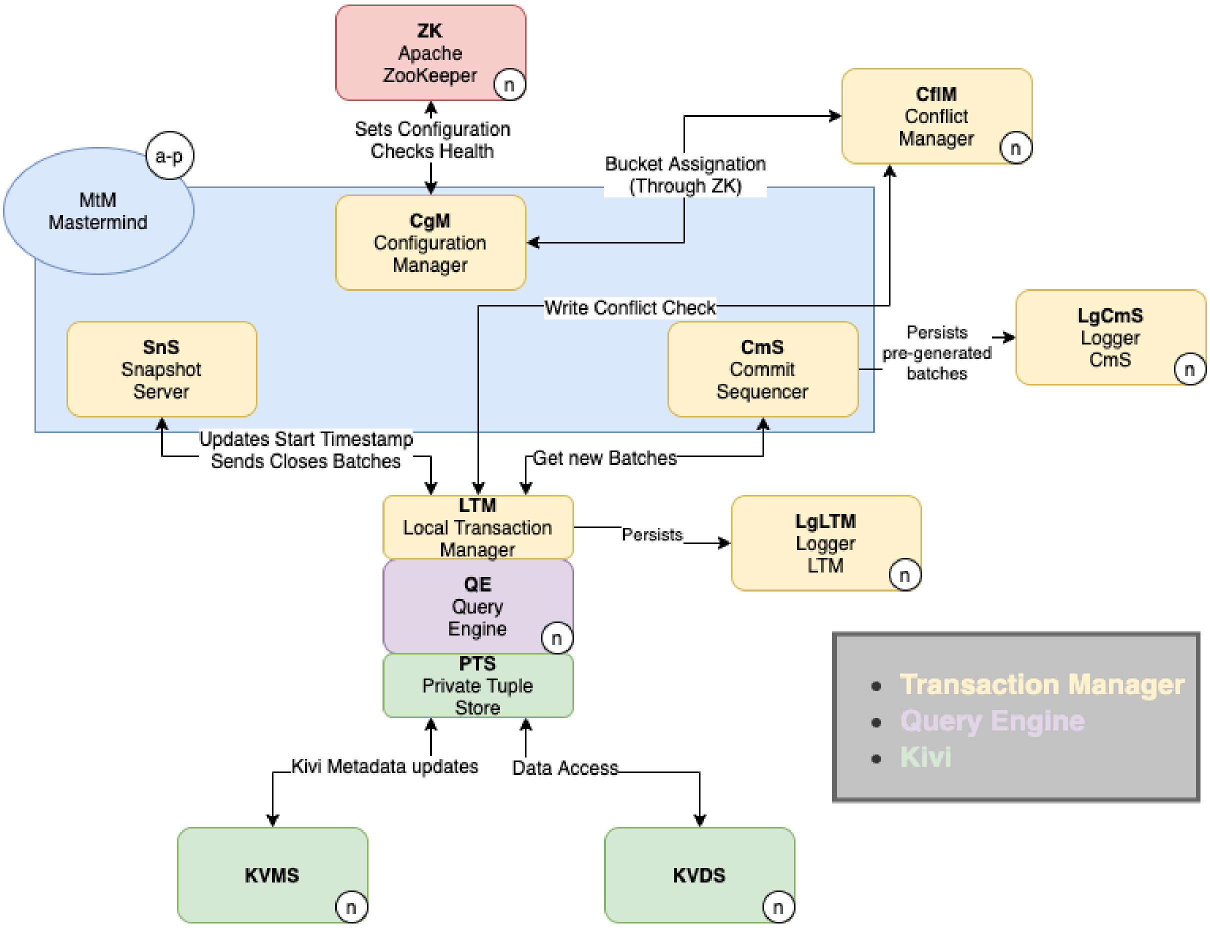


Figure 8: INFINITECH Data Management Deployment Diagram

From Figure 8 we can see that the majority of the components can be scale independently, apart from the configuration manager, the commit sequencer and the snapshot server. However, those components require very low volumes of CPU and memory usage and usually there is no need for them to scale horizontally. The configuration manager only holds information regarding the overall deployment and it is requested in case there is a need for a component to scale out, and the new deployed node needs to get information about the overall deployment in order to be able to connect to the other components. The snapshot server only receives notifications from committing transactions and does a small check to identify gap-free transactions in order to update the *snapshot timestamp*. Finally, the commit sequencer is responsible to increment the value of the *commit timestamp.* Due to the simplicity of these components by design, they have been unified in a single process that is called *MasterMind*. The *MasterMind* cannot scale out, however, in cases there is a need to server very high workload where the whole deployment consists of hundreds of nodes, we can scale up the instance by increasing the resources of the corresponding node.

On the other hand, the role of the conflict manager is to check for write-write conflicts in order to decide if an operation is allowed and which transaction should abort. It can be parallelized and scale out to many instances, thus it can support very high rates of operational workload. Each of those instances is responsible for a subset of data keys, which we refer as a bucket. Data keys consist of the concatenation of the unique table identifier plus the data keys themselves and are hashed and assigned to a bucket using the modulo function: bucket = hash(key) modulo with the number of overall buckets. The bucket is the unit of distribution for the conflict manager and each conflict manager is in charge of a number of buckets. Each bucket is handled by a single conflict manager.

The conflict manager keeps at most two values per data item: the commit timestamp of the last committed version and the start timestamp of an active transaction updating the data item, if any. This along with the distribution of conflict managers avoids the conflict manager being a bottleneck when it handles the whole set of keys, that can be huge. When a transaction Ti tries to modify a data element, it sends a request to the conflict manager to check for potential conflicts. This invocation is asynchronous, which means that it is not blocked until the conflict manager responds. In case of a conflict that must force the transaction to abort, this will be communicated in the next invocation of the conflict manager, or during the commit phase. This is irrelevant, as the transaction must be atomic, therefore, it does not matter which operation will cause the abortion, as all operations must be cancelled. A conflict is detected if the conflict manager has previously accepted a request from a concurrent transaction (either active or committed). Each transaction keeps how many conflict managers have been involved and upon successful completion of the commit phase, all the involved conflict managers are informed about its commit timestamp so that each conflict manager updates the information about the conflicts and can perform the proper checks for future transactions.

Regarding the *logging services*, these are responsible to ensure the durability of a transaction, which can be handled independently as well. The redo records of a transaction are pushed to the logging service and made durable before commit acknowledgement is returned to the user. The logging component, named logger, is parallelized and distributed by creating as many logger instances as needed to handle the required throughput. Each logger takes care of a fraction of log records. Loggers are totally independent, and they do not coordinate among them. Log records are inserted into the logger’s buffer. The buffer content is flushed at the maximum rate the underlying storage allows, minimizing the latency of logging.

3.2.3 Proactive Timestamp Management

As depicted in Figure 8, almost all components of the INFINITECH data management layer can scale out independently to hundreds of nodes in order to serve very high rate of operational workloads. An exception is the *MasterMind* process, which consists of the *configuration manager,* which does not need to scale out as it only holds information about the overall configuration is it is rarely invoked, and the *snapshot server* and *commit sequencer*. Given the overall transaction processing depicted in Figure 5, the request for getting the *start timestamp* is served by the snapshot server, while the request to commit a transaction is served by the commit sequencer. Internally, when the local transaction manager notifies that the private write-set is readable, the snapshot server is informed in order to update (if possible) the start timestamp, thus updating the visibility of the dataset. As discussed before, those two components are responsible for a tiny amount of work: increment their corresponding counter. However, even if the amount of work that they are responsible for requires very low amount of computational resources, serving very high rates of operational workloads would require their continuous invocation by the millions transactions and it might introduce a bottleneck due to the capacity of the network, apart from the inherit latency introduced by the invocation itself. As a result, the management of those two timestamps become the ultimate bottleneck of the system. We overcome this issue by implementing a proactive management of the timestamps, both at the snapshot server and the commit sequencer level.

Regarding the commit sequencer, it should be invoked once the transaction requests to commit and the conflict manager have already ensured that there are no write-write conflicts and the transaction can safely commit. The commit timestamp serves to tag the data version/snapshot with this logical number. As a result, if two concurrent transactions can safely commit, meaning there are no conflicts between them, it is irrelevant which one of the two will get the earliest number first. After all, the snapshot server will eventually update the visibility so that both of their modifications can be visible by forthcoming transactions. In fact, instead of having each transaction to communicate independently with the commit sequencer, the latter sends a batch of commit timestamps to the local transaction manager that is part of the query engine instance, and each transaction gets the commit timestamps immediately from it, thus delegating the task to synchronize with the commit sequencer to it in a separate thread.

The whole process is depicted in Figure 9. At the beginning, the commit sequencer is sending an initial batch of available timestamps to the local transactional manager. Transactions that need to commit in that instance of the query engine, will get values available from that batch. In that case, T1 will be assigned with commit timestamp 0 and T2 with the commit timestamp 1, without having to communicate with the commit sequencer itself. After a predefined period of time, the commit sequencer will send a new batch of available timestamps. During that period, no transaction requested to commit in that instance. It is important to notice that the query engine drops the previous batch and will server on-going transactions with values from the current active batch. As a result, when T3 needs to commit, instead of being assigned with value 2, it will get the first available value from the current batch, which is 3. Again, after the predefined period of time, the commit sequencer will send the new batch and the query engine will drop the current one. The reason for dropping old batches is to get synchronized with how timestamps are advancing globally. In a distributed environment with various instances of a query engine, the commit sequencer will send batches to each one of them. If some query engine processes transactions faster, it will receive larger timestamps while a slow query engine still uses low-value timestamps. This will delay the advancement of the global snapshot counter. As a result of discarding commit timestamps, the snapshot server is not only informed about the timestamps of committed transactions but also about unused commit timestamp ranges so that, it can advance the snapshot counter appropriately, without having to take into account the gab that is being created by the dropping of the batches.

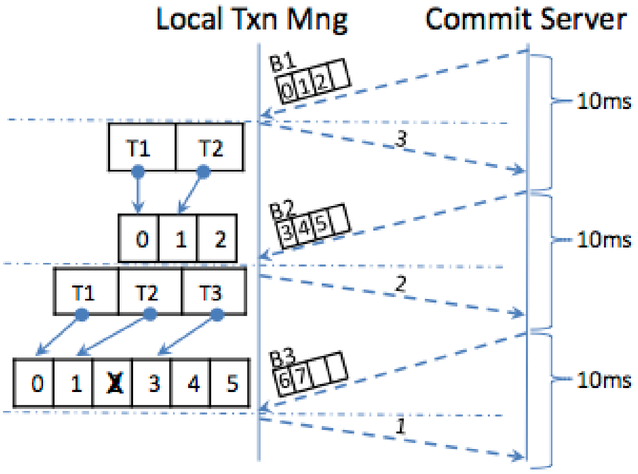


Figure 9: Proactive Commit Timestamp Management

When the commit sequencer sends the batch to the local instance, it gets informed about the number of overall transactions that have been committed during the previous period. By doing this, it can estimate the range of values that it should send to the query engine in order to avoid spending unused timestamps. In our example, we can see that after the second provision of the batch, it gets informed that only 2 transactions managed to commit, and it decreased the batch size of the third.

Regarding the snapshot server, it proactively reports the current start timestamp to each instance of the query engine by periodically send this value. As a result, we avoid the communication with the snapshot server on a per-transaction basis. Moreover, instead of sending the commit timestamps of each transaction to the snapshot server, in order for the latter to advance its counter and to forward the visibility accordingly, we batch those timestamps in groups and send them periodically. These are sent after the commit is informed back to the application and the flow is unblocked, so it does not have any impact on the overall latency of the transaction.

3.2.4 Asynchronous messages and batching

As implied by our implementation of the transaction processing, each transaction needs to send several messages from performing conflict detection, to read or write data and for logging in order to ensure the durability of the transaction. Those messages are synchronous, meaning that the process is blocked until a notification or a response is received, and as a result, they become part of a critical path that affects the overall response time of the transaction. In order to avoid these introduced latencies, we have adopted an asynchronous approach whenever possible. For example, instead of waiting to check for a write-write conflict, we send the request in an asynchronous manner and we unblock the operation, so that the transaction can continue without waiting for a confirmation by the conflict manager. If a conflict is detected, it will be notified to the transaction on the forthcoming request for a conflict. If there is no other request, it will be notified when the transaction requests for a commit. A transaction must be always atomic so it is irrelevant at which operation it will be notified to abort, as all operations must be cancelled.

Moreover, message exchanging for these operations also has a network cost and can also become critical in large-scale deployments, so there is the need to reduce the amount of messages as much as possible. We handle this issue by batching the requests and the responses extensively. In typical batching approaches latency is traded off for throughput what causes an increase of response times. We apply batching combined with asynchrony to avoid a negative impact on response time. During the conflict detection, a naïve implementation would be to send a request for conflict on each update operation. This will not only introduce a delay due to the round trip of message exchange but also will cost for CPU needed at the sending and receiving side. Instead, we keep track of all those requests and local transactional manager periodically sends a batch with all requests to the conflict manager. From the transaction point of view, it sends a request to its corresponding local manager for conflict detection and continues executing its next operation. The local manager periodically sends those batches, which are examined by the conflict manager. The latter sends back the responses with the results. By applying this technique, we don't affect the overall response time of the transactions, while at the same time we reduce adequately the amount of messages that need to be sent across the network. The increased latency of the batch exchange which contains numerous requests is being hidden by the concurrent execution. In fact, from a transaction perspective, it only needs to wait for the last batch to be received by its local manager.

3.2.5 Session Consistency

Our solution returns the commit to the client when durability is guaranteed but before the updates of the transaction are readable and visible. This approach might violate session consistency. That is, a client might not read its own writes across different transactions. Let us consider two consecutive transactions from one client, T1 and T2. T1 updates x and commits. T2 starts before T1's update is visible, and thus, receives a start timestamp smaller as T1's commit timestamp. Therefore, when T2 reads x it will not receive the version created by T1.

Session consistency can be implemented by delaying the start of transactions after the commit of an update transaction till the snapshot counter reflects the commit timestamp of the committed update transaction. Only then, the local transaction manager assigns the start timestamp to the transaction. This delay can amount up to a few tens of milliseconds that it is not an issue for OLTP response times that are in the range of 1-2 seconds. It should be noted that a client only pays this delay when it starts a transaction immediately after committing an update transaction. After committing a read-only transaction (typically around 90% of the transactions in OLTP workloads) no delay is paid. Also, if a client does something else between the commit of an update transaction and the start of a new transaction, this delay can be partially or fully masked.

4. INFINITECH OLAP Engine

The main goal of task T3.1 "Framework for Seamless Data Management and HTAP" is to provide the corresponding framework that will allow for seamless data access on data that might have been needed to be distributed among different datastores: an operational datastore that can ensure transactional semantics and a data warehouse that can be used to perform OLAP operations. Data are being copying to various locations which is cost expensive in terms of storage, while analytical operations are scanning a previous snapshot of the dataset, as the ETL operations that migrate data elements from an operational store to a data warehouse are being executed periodically. The INFINITECH solution provides a common platform that stores data and allows for both operational and analytical processing, without the need to copy data to different locations. This is achieved by providing HTAP capabilities, as explained in the previous sections. HTAP is feasible due to the use of the *snapshot isolation,* as presented in section 3. The ability to handle OLAP workloads and remove the need of migrating data to a data warehouse and delegate to the latter this processing, is based on the OLAP engine of the INFINITECH data management layer explained in this section.

The implementation of the OLAP engine is in progress at this phase of the project (M11) and as a result, at this version of the document we present the basic concepts that drive the overall design and implementation. This section will be further updated in the second version of the deliverable when the OLAP engine is planned to be delivered.

4.1 OLAP overview and connectivity

As depicted in Figure 7, the data management layer consists of three major components: the KiVi data store, the query engine and the transactional manager. The latter has been extensively presented in the previous section. Regarding the KiVi data store, it provides the persistent storage of the system. The data elements are stored in its data nodes, in a tabular format: It allows for key identification of a tuple, while a tuple can have various types of columns. It supports all standard SQL types, and additionally, it supports a column to be of a JSON type and enables query processing on the JSON, similar to MongoDB. Moreover, it can create indexes on specific columns, thus accelerating the data retrieval. It has been integrated with the *local transactional manager* that was presented in the previous section and therefore, it ensures transactional semantics and ACID properties. Finally, it exposes an internal API for query processing that can be used either directly by the application developer and data analyst, or by the query engine itself.

The Kivi Data Storage supports various operations for data modification and data retrieval, similar to standard SQL. It allows for data insertion, modification and deletion, while it supports data retrieval either by a *get* or a *scan* operation. The former can make use of the index and directly returns the corresponding tuple immediately, with a cost of O(1). A scan operation returns back a pointer to the first element that has been accessed, and using an iteration, it returns back the overall result set. It can benefit from the existence of an index to accelerate the process. It is important to notice that it can also support the ORDER BY operation, however only if the column to be ordered involves an index. It also provides filtering operations that allows the retrieval of a subset of a data table, according to the filtering properties. Finally, it also provides support for all aggregation operators supported by the SQL standard, such as *minimum, maximum, count, summary* and *average.* It is important to notice though that these operations are supported partially as they can be executed in a single data node. That means that if we have a distributed deployment involving several data nodes and a data table has been split among those nodes, the aggregate operations can be executed only per-node. The execution of the *minimum* operation will return the minimum value of a dataset in the specific node. It is up to the application developer and data analyst to retrieve the overall minimum of a dataset, by comparing the partial minimums that has been retrieved by each one of the nodes. Finally, it cannot support JOIN operations between tables.

Accessing directly the data nodes is feasible and accelerate the overall performance, as it avoids the inherit overhead introduced by the footprint of the query engine. However, the API is designed to for efficient data retrieval and therefore it does not comply with a specific standard, even if it is very similar to the one that MongoDB is offering. As the latter also does not comply with a specific standard but instead it customized to match the specific needs of the MongoDB datastore, in a similar way, the API that the data storage is providing also is designed to match the specific characteristics of the storage layer itself. As a result, the direct API cannot be integrated with the popular analytical tools that are dominant in the insurance and finance sectors, such as Apache Spark[5] or Apache Hive[6].

In order to overcome this issue, there has been implemented a series of additional components that can be used instead. Firstly, a python implementation of the driver is provided. Data analysts that rely on python programming language in order to write scripts and feed their ML/DL algorithms can benefit from it, as it exposes native python methods that can be used by the analyst. This driver consists of a wrapper that encapsulates the complexity of the underlying methods and the connectivity details of the API. Moreover, in order to be compliant with popular analytical frameworks often used by analysts in the insurance and finance sectors, there has been provided an implementation of the OData specification[7]. The latter is an OASIS standard that defines a REST API in order to access data in data management system. It provides functionalities for data retrieval and data modification. It also defines a pair of basic web methods that allows the analyst to execute data aggregation operations. As this specification is an OASIS standard, it can be effectively integrated with a variety of analytical frameworks that already are compatible with this standard.

Additionally to the KiVi data storage element, there is also an implementation of the query engine, which is based on the Apache Calcite framework[8]. The latter has been extended to support DDL scripts and data modification operations. It provides a standard Java DataBase Connectivity (JDBC) driver and its dialect has been extended in order to provide all standard SQL operations, such as data modification operations (e.g., INSERT, UPDATE, DELETE) that were missing by the framework. The query engine is also integrated with the *local transactional manager* in order to ensure transactions and makes use of the direct API of the KiVi Data Storage for data accessing. By using the query engine via its JDBC driver the application developer and data analyst can benefit as the driver can be directly integrated with all popular analytical frameworks that can push down the query execution directly to the datastore. Being compatible with the standard SQL makes it a powerful tool for data processing, as the engineer and analyst can write complicate statements and delegate the query engine itself to process the data. Its query optimizer allows to transform the input statement into an equivalent one that can accelerate the overall response time of the execution.

It is important to mention that the query engine does not implement the OData specification, and it is not planned to do so, as it will be meaningless. The OData REST API will provide a third element on

[5] https://spark.apache.org/

[6] https://hive.apache.org/

[7] https://www.odata.org

[8] https://calcite.apache.org/

top of the data storage to retrieve data that it is totally unnecessary. It only defines a small subset of data operations so there is no operation that cannot be used directly via the JDBC. As a result, it provides no benefit to implement this standard on top of the query engine. In cases an analytical framework or other micro service requires the use of REST, it can use the implementation that directly access the KiVi Data Storage. In fact, using a REST API will imply that complicated operations must be done in the application level and cannot be pushed down. This makes the use of the query engine meaningless, as its purpose is to provide SQL support and optimize the query execution.

4.2 Query Optimization

The OLAP core is part of the query engine and consists of various components that interact in order to facilitate and optimize the execution of a statement. The most important of those are the following:

- Query Planner: It receives a query statement and transforms it equivalent ones, in the sense that the execution of each one of those *plans* will return the exact same result set.
- Query Optimizer: It receives a list of equivalent plans, estimates their cost and decides which of the proposed ones will have the minimum performance value.
- Query Executor: it receives the plan decided by the query optimizer and establishes the data pipeline needed for the query execution and data retrieval.

When the query planner receives a statement, it uses an internal compiler in order to create a structural representation of the script. Each part of the script involves a specific *query operator* and the compiler creates a structural tree which connects all the operators that need to take place in order to execute the query. Let's have a look at the following query:

```
SELECT t1.name, t2.account_number
FROM Persons as t1 INNER JOIN Accounts as t2 on t1.person_id = t2.person_id
WHERE t1.age > 60
```

This defines a *scan* operation on two tables, a *join* operation over two tables, a *filtering* operation with a specific predicate, and a *projection* over two fields. The tree of the query operations of this statement will be the one depicted in Figure 10.

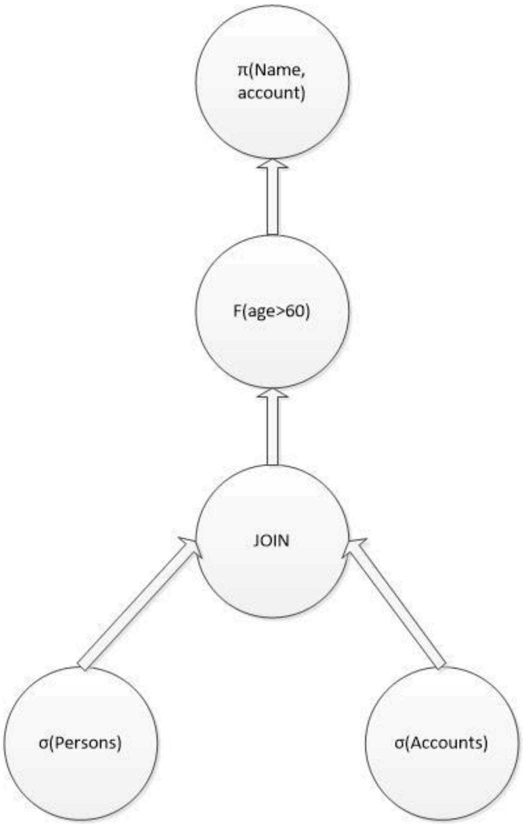


Figure 10: Tree of query operators

We can see that this query will execute two scan operator over the *Persons* and *Accounts* data tables, will join the intermediate results, then will filter out those tuples that have the age column lesser than 60, and will finally remove all other columns apart from the *name* and *account* that need to be projected to the final result set.

When the query planner constructs the tree of query operations, it applies various transformation rules that are available in order to propose equivalent plans. The rules should result to a plan that is valid and equivalent. For instance, an alternative plan can be the one depicted in Figure 11.

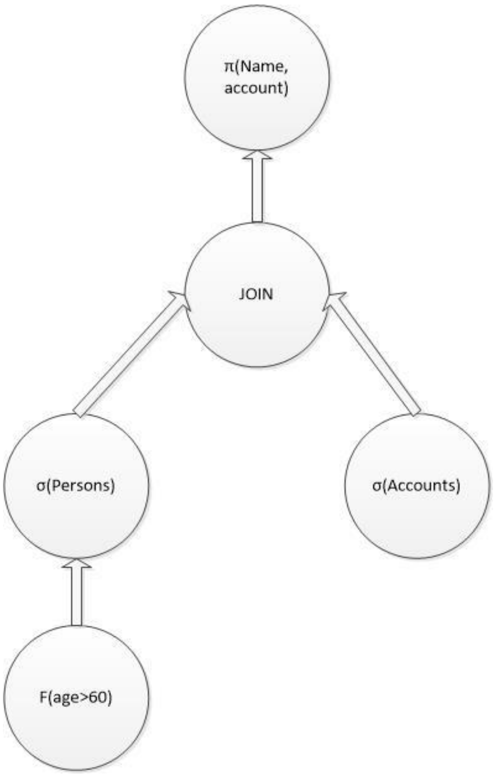


Figure 11: Alternative query plan

The planner proposes an alternative plan that the *filtering* operation should be pushed down and executed before the selection of the tuples of the *Person* table. Making use of a greedy algorithm, it produces various transformations that are being sent to the query optimizer to estimate the cost of each of the plans. The cost is taking into account three metrics: number of I/O access, size of tuple and number of rows each operator will return. The number of I/O access is important as this is a heavy operation with an increased latency. The size of the tuples affects the number of bytes that need to be sent over the network when fetching data from the data storage and the size of memory the query engine requires in order to execute the statement. Finally, the number of rows is important in operations such as the various implementations of the *join* operation or *ordering* when this cannot be pushed down to the KiVi data storage level. The optimizer makes use of various statistics available by the latter in order to have a better estimation of the cost. For instance, it can know if a column is indexed, the histogram of the distribution of the data over the index etc.

Each type of operator might have various implementations. For instance, the *filtering* operation might have an implementation that sends the filtering to be executing in the KiVi data storage level or in the query engine itself. For instance, Figure 10 implies a *filtering* implementation on the query engine, while Figure 11 implies that the filtering will be pushed down to the KiVi data storage. As the latter provides support for the majority of the SQL operations, many operations can be pushed down to that level, accelerating even more the overall performance by exploiting data locality and the fact that lesser tuples are being transmitted over the network and lesser number of tuples need to be processed in the query engine level.

The *join* operation is another example of an operation that has various implementations. Currently, the query engine has been designed to support the *nested-loop join,* the *merge join,* the *hash join,* the *equity join* and the *bind join.* Apart of the *nested-loop join* operations, all others are currently either under implementation or under evaluation by the query optimizer, and therefore more information regarding the details of the different implementations will be given in the next version of the deliverable.

Another important feature of the query engine is the ability to create custom operations via the use of *table functions.* This can extend the already functionality and enable it to access data from external sources. For instance, an implementation of a *table function* might allow executing a query statement in MongoDB and retrieving the results to the query engine. This implementation of the *table function* will be part of the tree of operators that the planner proposes and will be part of the data pipeline established by the query executor. By doing this, we can enable *polyglot* capabilities in the INFINITECH data management platform. These will be explained in the corresponding series of deliverables of task T3.2 "Polyglot Persistence over BigData, IoT and Open Data Sources".

In our example, the most efficient query plan will be probably the following:

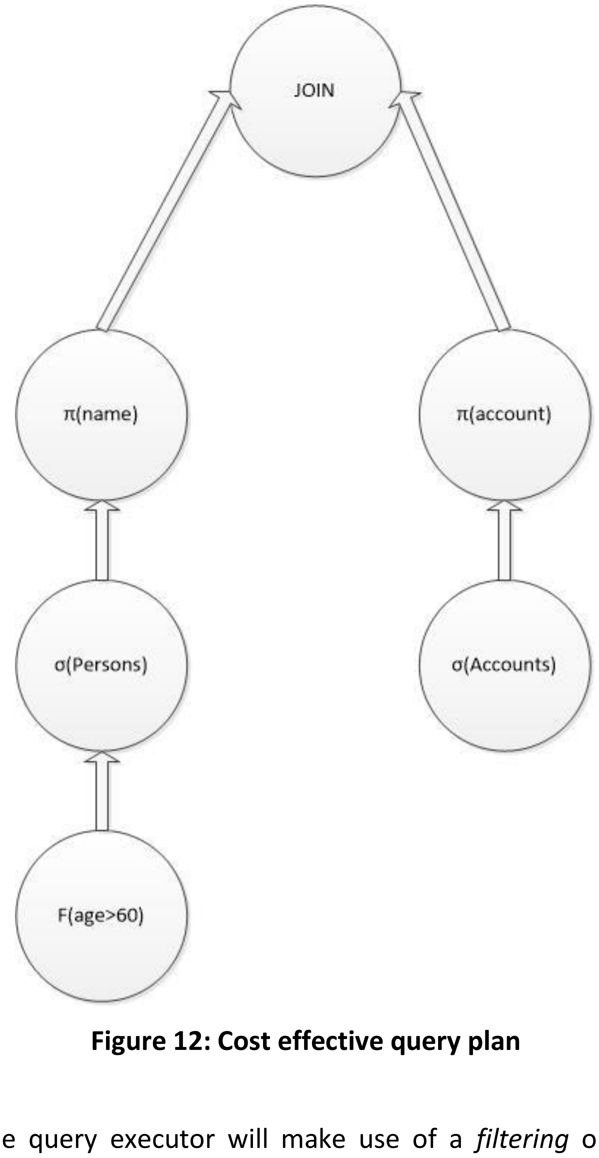


Figure 12: Cost effective query plan

This will implies that the query executor will make use of a *filtering* operation in the Kivi Data Storage level to eliminate the tuples of the *Persons* table that won't be part of the JOIN and do not need to be transferred to the query engine level, it will push down the *projections* to the Kivi Data Storage again, so as to reduce the amount of data that need to be transmitted over the network, and finally, it will make use of a JOIN operation in the query engine level. According to the information available by the KiVi data storage, it can use different types of implementations of the

JOIN. For instance, if the column *person_id* of the data table *Accounts* is indexed, this means that it is possible to retrieve data in an ordered manner on this field. Assuming that the *person_is* of the *Persons* table is a primary key, and therefore indexed by default, both selections in the KiVi Data Storage can retrieve data in an ordered fashion. As a result, the *merge join* implementation might be the optimal, as it quickly eliminates data that won't be part of the final result (i.e. tuples concerning accounts that are related with people younger than 60 years old), but it requires that both operands of the JOIN must be ordered. At the end, the query executor initializes the instance of all those operators, establishes a data pipeline and starts asking for data from the top operator. All operators implement the same interface, so that the details of the execution are transparent regardless of the type the operator. Each time, the executor invokes the *next* method of the interface, and its implementation asks the operator above according to the pipeline to get more data, by also invoking its *next* method. Operators also might pre-fetch data in order to accelerate the data movement across the different nodes of the query plan.

4.3 Parallel OLAP Engine

As mentioned at the beginning of this section, the main objective of Task T3.1 is to provide a data framework for seamless data accessing when mixing OLTP with OLAP workloads. Typically, for the former, operational datastores are being used that are capable to ensure transactional semantics, while for the latter, data are being migrated to data warehouses that provide powerful analytical capabilities and can process large amounts of data efficiently. The INFINITECH data management platform intends to covert this need for HTAP processing. We've already explained how the system can handle very efficiently OLTP workloads and how it can be used for data retrieval supporting all types of analytical operations. The previous subsection described how it can benefit by the query optimizer of the query engine to accelerate the response time of the query execution as it can propose an optimal execution plan. However, in order to truly fulfil the requirements of an OLAP datastore that has been built to efficiently execute analytical queries, the data management engine must allow for the parallel execution of the query. In INFINITECH, we provide 4 types of parallelism:

- Inter-query parallelism: Each query can be executed in a different node. The end-user submits a statement via the JDBC driver, and the latter decides which of the available instances of the query engine has lesser amount of work and consumes lesser resources, so that it can assign the execution of the query to a specific instance.
- Intra-query parallelism: The query itself can be split and can be executed in parallel in different nodes in a distributed manner.
- Inter-operator parallelism: Each operator that is part of the query can be executed by a different node. That way, different nodes can be responsible for the execution of different operators and thus, distribute the computational resources needed among the instances of the query engine.
- Intra-operator parallelism: An operator itself can be executed in a distributed manner. This means that it can be split into different data nodes and the master node can merge the partial results and return the overall set to the upper layer in the query plan.

The innovation of the INFINITECH data management system with respect to its analytical capabilities lies in its ability to provide intra-operator parallelism to cost demanding operators, such as the various aggregations and the *join* operator. In fact, the ability of the KiVi Data Storage to partially execute aggregations together with the intra-operator parallelism, give a lot of improvement in the overall response time of the execution. The whole operator can be pushed down to the KiVi Data Storage and can be executed in a distributed way. An aggregation operator typically involves a scan operation that retrieves the tuples that need to be calculated and then applies the aggregation. In a distributed deployment where a data table has been split to various data nodes, each one of the nodes will only scan the amount of data that is concerned, and therefore the overall execution can be benefit from the parallel scan that takes place on each of the data nodes. After the scan, the aggregation is applied and the results are merged from each of the partial executions. To clarify how this happens, the overall *minimum* value is the *minimum* value of the partial results. The overall *maximum* value is again the *maximum* value of the partial maximums. The overall *count* and *summary* are also the summary of the partial *count* and *summary* results. The only difference comes with the overall *average* which cannot be the average of the partial *average* retrieved by each data nodes. However, the overall *average* is defined by the overall *summary* divided by the overall *count*. As we saw, those two operations can be executed in a distributed manner and therefore, the overall *average* can be also executed distributing the load to different nodes.

In order for the intra-query and intra-operator parallelism to be achieved, it is required that a *data shuffle* operator is available. This is important for the *join* operator that cannot be pushed down to the KiVi Data Storage, as the latter does not support this operation. The *shuffle operation*broadcasts data retrieved by the distributed operators to all involved nodes. That way, in case we have *join* between two tables that will be implemented by a *nested-loop join*, this will involve the *scan* of the data of each of the tables. The OLAP engine can push down the *scan* down to the KiVi Storage, and the operation can be executed in a distributed manner, as shown. As we support intra-operator parallelism, the join will be executed in a distributed manner among the instances of the query engine. The master node fetches data from the left side of the *join,* those are being broadcasted to all nodes that are executing the operator. Each of those then picks up the value from the pipeline of the shuffle and enforces the logic implemented by the operator to retrieve only the corresponding value of the right hand of the join. The intermediate results are being collected by the master node which in turn, returns back the result to the upper layer of the query plan.

5. Conclusions and next steps

This document reported the work that has been currently done in the scope of task T3.1 “Framework for Seamless Data Management and HTAP” whose main objective is to provide a seamless way for data management across operational and analytical data stores by supporting the Hybrid Transactional and Analytical Processing (HTAP). The result of this task consists of the core INFINITECH data management platform that is a central data repository that can store data and provides a seamless way for OLTP and OLAP operations, without the need to migrate data from an operational datastore to an analytical store using expensing ETLs for data migration. This eliminates the need for keeping multiple copies of data over various types of stores and additionally allows for analytical processing over live data, and over a previous snapshot of the dataset that was taken at the time the ETL was executed.

This report provides an analysis over the read phenomena that occur according to the isolation level and concludes that the organizations in the finance and insurance sectors require a higher level isolation, which will downgrade the level of parallelism of the concurrent execution of transactions. We decided that the traditional *two-phase locking* mechanism that is implemented by traditional operational data management systems will create a bottleneck when mixing operational with analytical workloads and we saw the benefits of relying on the *snapshot isolation* paradigm for building our transaction engine.

Based on this decision, we designed the INFNIITECH transactional engine in a manner that is fully scalable in order to handle very high rates of transactions. Insurance and finance institutions are expecting very high rates of traffic as they have to serve millions of finance transactions per minute or need to take into account millions of sensor IoT data. Instead of a monolithic approach, the transactional manager, as presented in section 3, can scale out its components independently in order to handle those high rates. In the cases that a bottleneck can occur due to the fact that some components cannot scale out, a proactive approach and an asynchronous communication have been applied that eliminates the issue. Therefore, the data management layer of INFINITECH is expected to perform better than the traditional operational datastores, as it solves the issues and bottlenecks that those stores impose when dealing with the high OLTP workloads that the finance and insurance organizations produce.

Moreover, its OLAP engine provides all standard SQL capabilities that typical analytical datastores provides and it is compatible both with the OData standard, exposing a REST API, and implements the JDBC specification for data connectivity. By doing this, it can be integrated with all popular analytical frameworks used by the data analysts for ML/DL activities in the finance and insurance sectors. Its engine supports query optimization by transforming the query in order to take advantage of the characteristics of its internal storage engine, while it supports the parallel execution of the query statement, in order to achieve the same performance as typical analytical datastores. This, combined with the ability to perform OLTP workloads on the same data, eliminates the need for the migration of the operational data to a data warehouse using expensive ETLs. This enables organizations in those sectors to perform effectively analysis over the real data, thus providing real-time business intelligence to their customers.

To conclude, the progress of the task T3.1 is in plan with the timeline, and an initial implementation has been provided. At this phase of the project, the HTAP capabilities have been supported by the transactional processing that the INFINITECH data management platform offers. Currently, the OLAP engine is extended in order to improve its internal query optimizer for taking advantage of the unique characteristics of the storage level, while the parallel processing is under implementation that will improve the overall performance of the analytical queries. In the second and third iterations of this deliverable, additional details regarding the OLAP engine will be provided, along with some benchmarks based on the TPC-* family of frameworks. This will validate the INFINITECH solution over other solutions used in the finance and insurance sectors for supporting transactional and analytical processing.