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

Infinitech

D3.10 – Automatic Parallelization of Data Streams and Intelligent Pipelining - II

Revision Number	3.0
Task Reference	T3.4
Lead Beneficiary	LXS
Responsible	Ricardo Jiménez-Peris
Partners	LXS, GLA, UNP
Deliverable Type	Report (R)
Dissemination Level	Public (PU)
Due Date	2021-07-31
Delivered Date	2021-07-27
Internal Reviewers	NUIG, UPRC
Quality Assurance	INNOV
Acceptance	WP Leader Accepted and Coordinator Accepted
EC Project Officer	Pierre-Paul Sondag
Programme	HORIZON 2020 - ICT-11-2018
	This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement no 856632

Contributing Partners

Partner Acronym	Role[1]	Author(s)[2]
LXS	Lead Beneficiary	Ricardo Jiménez-Peris
LXS	Contributor	Boyan Kolev, Pavlos Kranas, Spencer Pablos, Patricio Martinez, José María Zaragoza, Jesús Manuel Gallego
GLA	Contributor	Richard Mccreadie
UNP	Contributor	Bruno Almeida Tiago Teixeira
NUIG	Internal Reviewer	Martin Serrano
UPRC	Internal Reviewer	Dimitris Kotios
INNOV	Quality Assurance	Dimitris Drakoulis

Revision History

Version	Date	Partner(s)	Description
0.1	2021-07-09	LXS	ToC Version and updated initial input of D3.10
0.2	2021-07-09	All	Section 5 added
0.3	2021-07-22	LXS	Update Executive summary, intro and conclusions
1.0	2021-07-22	LXS	Submitted for internal review
1.1	2021-07-26	UPRC	Internal review
1.2	2021-07-26	NUIG	Internal review
2.0	2021-07-26	LXS, GLA	Submitted for QA
3.0	2021-07-27	LXS	Finalized the document

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

[2] Can be left void

Executive Summary

The goal of task T3.4 “Automated Parallelization of Data Streams and Intelligent Data Pipelining” is to firstly provide the INFINITECH approach for intelligent data pipelines and secondly, to allow for the automation of the deployment of parallelized data streams. Modern applications currently being used by data-driven organizations, such as those belonging to the finance and insurance sector, require to process data streams along with data persistently stored in a data base. A key requirement for such organizations is that the processing must take place in real-time providing real-time results, alerts or notifications in order to instantly detect fraud financial transactions the moment they are being occurred, detect possible indications for money laundering or provide real-time risk assessment among other needs. Towards this direction, streaming processing frameworks have been used during the last decade in order to process streaming data coming from various sources, in combination with data persistently stored in a database that can be considered as data *at-rest.* However, processing data *at-rest* introduces an inherit significant latency, as data access involves expensive I/O operations, which are not suitable for streaming processing. Due to this, various architectural designs have been proposed and are utilized in the modern landscape that deals with such problems. They tend to formulate data pipelines, moving data from different sources to other data management systems, in order to allow efficient processing in real-time. However, they are far from being considered as *intelligent*, with each of the proposed approaches comes with their own barriers and drawbacks.

A second key requirement for data-driven organizations in finance and insurance sector is to be able to cope with diverse workloads and continuously provide results in real-time even when there is a burst of incoming data load from a stream. This might occur in case of having a stream consuming data feeds from social media in order to perform a sentiment analysis and an important event or incident takes place, which will make the social community response by posting an increased number of tweets or articles. Another example is the unexpected currency devaluation that will most likely trigger numerous financial transactions caused by people and organizations swapping their portfolio currencies. The problem with the current landscape is that modern streaming processing frameworks allow for static deployments of data streams that consist of several operators, in order to serve an expected input workload. In case of such scenarios, an unexpected burst of the incoming workload might saturate the resources devoted for the initial deployment which cannot provide the results in real-time, or even worse, might lead to a crash.

In order to cope with the abovementioned requirements and overcome the current barriers of the modern landscape, we envision the INFINITECH approach for Intelligent Data Pipelines and the parallelized data stream processing using Apache Flink as the baseline technology for the INFINITECH streaming processing framework along with Kubernetes. In our solution we provide a holistic approach for data pipelines that makes use of the key innovations and technologies provided by INFINITECH and implemented in the rest of the tasks of the project related with data management activities. Our solution solves all problems related with different types of storage and the usage of different types of databases for persistent data storage, allowing efficient query processing, handling aggregates and dealing with snapshots of data. Moreover, we have designed our solution for parallelized data stream processing, allowing the deployed operators to save and restore their state and the online reconfiguration of the Flink clusters, which enables elastic scalability by programmatically scaling the clusters.

This document reports on the work that has been done towards those two main objectives at the current phase of the project. An initial state-of-the-art analysis of the current status of the data pipelines and data stream parallelization has been provided, along with the discovered barriers, problems and solutions used so far. Then, we provide the current landscape of data pipelining in modern enterprises today, analyzing the different architectures used to cope with the inherit challenge of combining streaming data with data *at-rest*, along with each solution’s benefits and drawbacks .We then defend on how the INFINITECH approach can be used to solve those issues in a holistic manner, utilizing the development of the other tasks and the INFINISTORE as its basic pillars. At this second phase, we went one step beyond, making use of the *change data capture* paradigm for implementing the data pipelines. We provide a demonstrator on how we can move data from external operational datastore to INFINITECH in a transparent way. Then, we

describe the initial design of our solution which allows the dynamic redeployment of the parallelized data streams to enable the elastic scalability of the deployed operators. In the final version of the document, we provide implementation details regarding these parallelized data streams and extend the data pipelines to enable other targets to consume data from INFINISTORE. This will be of significant value for the Semantic Interoperability Engine of INFINITECH.

Table of Contents

List of Figures

Abbreviations/Acronyms

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
CDC	Change Data Capture
CEP	Complex Event Processing
CPU	Central Processing Unit
DoA	Description of Action
DNS	Domain Name System
ELT	Extract, Load, Transform
ETL	Extract, Transform, Load
FP7	7th Framework Program
HDFS	Hadoop Distributed FileSystem
HTAP	Hybrid Transactional and Analytical Processing
I/O	Input / Output
INFINISTORE	The INFINITECH data management layer based on the LeanXcale database
IoT	Internet of Things
JSON	Javascript Object Notation
MoM	Monitor of Monitors
OLAP	Online Analytical Processing
SQL	Structured Query Language
WP	Work Package
XML	Extensible Markup Language

1 Introduction

Data-driven organizations nowadays are increasingly requiring the combination of streaming data with persistently stored data, usually called data *in-flight* and data *at-rest*. Streaming processing frameworks that have been adopted and widely used during the last decade are being enhanced with additional functionalities for solving the inherit barriers that database management systems introduce to an integrated solution. The most important obstacles faced are the latency of an analytical query processing operation over a persistent data store and that datastores are not designed to support data ingestion in very high rates in order to serve an increased data load coming from a stream. Due to this, they tend to use a variety of different data management systems, from operational datastores, to data warehouses and data lakes. Operational datastores allow to persistently write data, but are not appropriate to perform analytical query processing over bigdata. For this purpose, data is being periodically moved to data warehouses that are designed to only allow read operations, using sophisticated indices and data structures that can boost the performance of such operations. Data lakes can also be used as a cost effective solution to store historical data that does not require frequent processing. Operational datastore on the other hand can be further divided to different categories: traditional SQL datastores that ensures consistency in terms of database transactions, which are critical for applications in the insurance and finance sector. However, they are not designed to scale out, and they are inefficient in performing analytics on the same time. For this reason, other types of operational datastores have been adapted during the last decade, commonly referred as NoSQL datastores (or their evolutions which are widely known as NewSQL), which sacrifices transactional semantics for the sake of scalability. Usually, they can scale out more efficiently and can serve ingestions in very high rates. However, they lack rich query processing capabilities. As a result, modern enterprises use a variety of different datastores, stream data managers and tools, such as Kafka and machine learning infrastructure. This makes the data pipelines more complex and problematic, as they need to move data across the different databases, in order to take advantage of each one's benefits. For that, they rely on expensive ETLs (Extract, Transform, Load) and they perform periodic batch processing, which is not suitable when there is the need for real-time data analytics. Periodic batch processing makes the data used in a processing framework obsolete, as ETLs take place periodically, once every day or during weekends.

In order to overcome these problems, different architecture designs have been proposed and adapted in modern enterprises. For instance, lambda architectures have been widely adopted to solve the problems of complexity mixing different databases and the need for real-time processing. But they are very complex, consisting of different layers with different codebase, while their maintenance is hard to keep. Other architectures rely on moving data from operational to analytical datastores and vice-versa, using architectural designs such as *current-historical data splitting*, *data warehouse or operational data offloading* and *database sharding*. All of these come with the drawback that query processing takes place over a snapshot of the dataset, and the results are obsolete. Other approaches aim at improving the latency of the execution of an analytical query, which involves aggregations. This is crucial as the response time must be very low in order to be used in combination with streaming operators. *Detail-aggregate view splitting, in-memory application aggregations* and *federated aggregations* are techniques widely used to solve these issues, with the drawback of sacrificing the consistency and accuracy of the results. We provide details on these designs in section 3 of this report.

To solve these issues, we propose the INFINITECH approach for Intelligent Data pipelines, which provides a holistic solution for data pipelines, solving major problems with different types of storage, handling of aggregates and dealing with snapshot databases. Our proposed analytical pipeline will address all of the above identified architectural patterns for data pipelining, combining data streaming and data at rest, taking advantage of the technologies and innovations developed in INFINITECH that break through the current barriers of modern applications in finance and insurance sectors which have been reported so far in the corresponding deliverables such as D3.1, D3.4, D3.6, D5.1 and D5.4 that report the work that has been carried out so far in the rest of the tasks related with the data management layer of IFINITECH. We will build our solution taking these prototypes as the basic pillars.

Another key requirement for modern data-driven organizations using streaming processing frameworks is the ability to cope with diverse incoming workloads. So far, current solutions allow only static deployments of data stream operators. This is problematic in the sense that in cases of unexpected peaks of incoming data coming from a stream, the static deployment cannot scale out to cope with that need.

In order to help solving the abovementioned issues and according to the various INFINITECH pilots, we propose a solution based on Apache Flink as the streaming processing framework of INFINITECH, integrated with the INFINISTORE as the data management layer and in combination with Kubernetes as the container-orchestration framework. By using INFINISTORE, we can cope with the majority of technological challenges for data pipelines that require complex architectures that introduce additional barriers, as explained in section 4 of this report. The use of Flink allows us to also extend their operators to store and restore their state using checkpoints. By doing so, we are now capable of redeploying a Flink cluster, increasing the number of instances of the operators and restoring their relevant state. This allows programmatically scaling the Flink clusters and providing dynamic redeployments in order to cope with these diverse workloads.

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 T3.4 “Automated Parallelization of Data Streams and Intelligent Data Pipelining” at this phase of the project. This task lasts until M30 and therefore, two additional versions will be released, extending and modifying the content of this document. During this phase, the identification of the problem that needs to be solved took place, by performing a thoughtful state-of-the-art analysis on problems and solutions of data pipelines and data stream parallelization, along with a detailed analysis of their current landscape in modern enterprises of today. We identify which different technologies are mostly used in order to combine processing of streaming data with data at-rest in a data pipeline, and the dominant architectural designs used so far, in order to identify the current drawbacks. We then propose the INFINITECH approach for data pipelines, that will solve those problems in an holistic manner. Moreover, we designed our solution that allows for dynamic redeployments of streaming operators. This will allow for a dynamic scaling of those operators, which is a current challenge.

In the next iterations of this deliverable, a more detailed description of our implementation will take place, along with the experimentation with use case scenarios coming from the pilots of INFINITECH. However, our solution is case agnostic, and has been designed to fit any other user story.

1.2. Insights from other Tasks and Deliverables

The work that is reported in this deliverable is based on the overview of baseline technologies defined in WP2. This task is based on the technologies and innovations implemented under the scope of the tasks of the project that are related with the data management layer and make use of those at the basic pillars. As a result, is very closely related with T3.1 “Framework for Seamless Data Management and HTAP”, which provides the fundamentals that allows the hybrid transactional and analytical processing, allowing the for query processing over live data added in the operational datastore, removing the need to migrate data to a data warehouse. Moreover, T3.1 allows for the direct data ingestion in very high rates, which removes this barrier from our solution. T3.2 “Polyglot Persistence over BigData, IoT and Open Data Sources” provides the polyglot extensions in the level of INFINISTORE query engine that allows for a unified manner to query data that are stored in a different data sources. T5.3 “Declarative Real-Time Data Analytics” implements the *online aggregates* that will be massively used in our solution, removing the inherit barriers of data consistency that various architectural designs suffers when trying to pre-calculate the results of the analytical operations in order to boost the performance of such operations. Finally, T3.3 “Integrated

Querying of Streaming Data and Data at Rest” provides the Apache Flink as the streaming query processing framework of INFINITECH, integrated with the data management layer (the INFINISTORE) to allow the combination of streaming processing with data *at-rest* in INFINISTORE. With this integration, we are now capable of taking advantage of all these aforementioned technologies into our INFINITECH approach for Intelligent Data Pipelines. Last but not least, the INFINITECH way for deployment, using tailored sandboxes that rely on the Kubernetes, as implemented under the scope of WP6 “Tailored Sandboxes and Testbeds for Experimentation and Validation” and the provision of the reference testbed, in combination with the use of Apache Flink as the baseline technology for streaming processing, allows for the parallelization of the data streams which is the second important objective of the task.

1.3. Updates from the previous version (D3.9)

In this version of the report, we have added section 5 which provides a demonstrator of how an INFINITECH Data Pipeline can be configured, deployed and executed. We firstly introduce the *change data capture* paradigm which uncovers the basic principle behind our implementation, along with the *Debezium* framework that will be used as the background technology. Then, we provide a demonstrator for a data pipeline between an operational datastore that lies outside an INFINITECH deployed sandbox and the INFINISTORE. In the next subsections, we give more details with a hands-on about configuring this data pipeline and deploying it. Then, we simulate the scenario where data is being ingested in the operational datastore and transparently moved to the INFINISTORE, whose innovation features can be further exploited by data analysts or application developers, removing the need for the complex hybrid architectures that has been explained in detail in section 4. Finally, we give an overview of the next steps and planning for the final phase of this task.

1.4. 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 the tasks related with the data management activities of the project. Section 2 provides an extensive state-of-the-art analysis on data stream parallelization, describing the problems and the most adapted solutions. Section 3 describes the current landscape of modern applications that make use of the data pipelines while section 4 introduces our vision for the INFINITECH approach for Intelligent Data Pipelines, after providing an analytical survey of modern architectural designs along with their drawbacks and inherit issues, describing how our approach can provide a holistic way to solve all those issues. Section 6 provides a demonstrator regarding the implementation of an INFINITECH data pipeline that moves transparently data from an operational data store to the INFINISTOE. Section 6 deals with the problem of dynamic reconfiguration of streaming operators that will allow dynamic scalability and also describes the design of our solution. Finally, section 7 concludes the document and presents the next steps towards the delivery of our prototype.

2 State-of-the-Art Analysis on Data Stream Parallelization

2.1 Introduction

There is an increasing demand in data-driven organizations to process data streams as opposed to only stored data. A data stream is a sequence of tuples with some pre-defined schema. The main difference with a database is that a database query processes a snapshot of data at a particular point in time and produces the answer, while a streaming query is continuous and produces results continuously. Basically, a stream comes from a data source and contains tuples. A data streaming query processes this continuous sequence of tuples producing a continuous stream of results. Data streaming has many applications in the financial and insurance world such as fraud detection, IoT, stock trading, etc. For this reason, they are becoming more and more important in data-driven organizations.

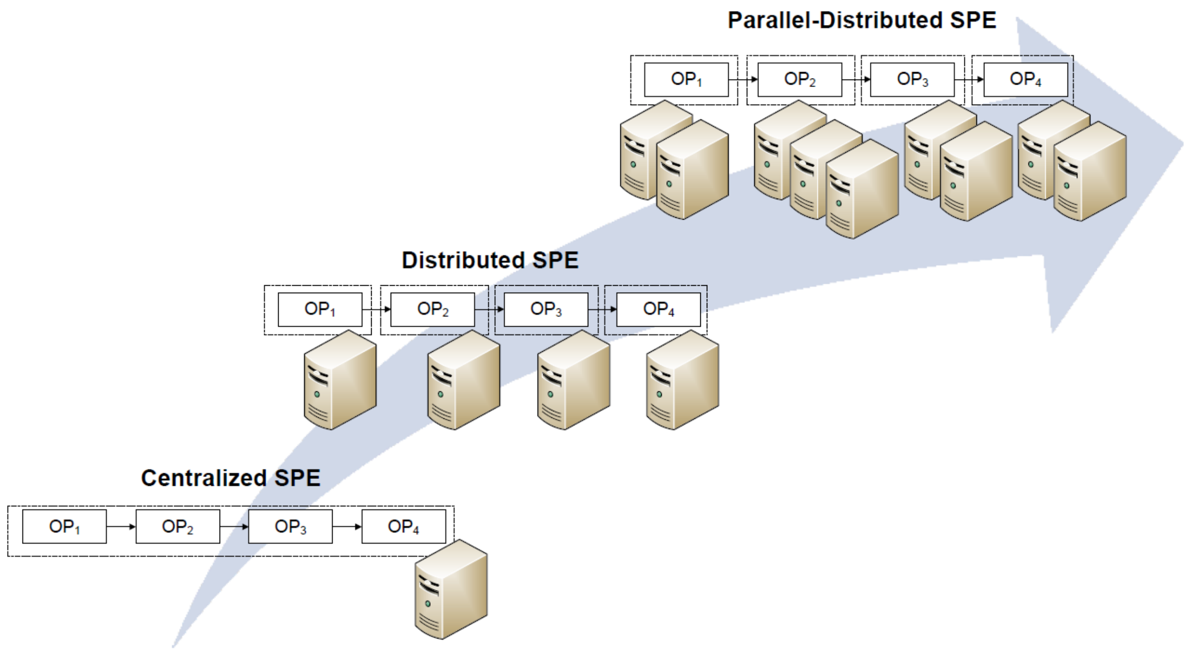

Figure 1: Stream Processing Engines Evolution

Data streaming engines started as centralized systems such as NiagaraCQ (J. Chen, 2000) and TelegraphCQ (S. Chandrasekaran, 2003), or Aurora (D. Carney, 2002). The main issue with centralized engines is that they are limited to the capacity of a single node and they cannot process large stream volumes. For this reason, distributed stream processing systems started to be built introducing different kinds of parallelism in the query processing. In the next section we elaborate on the different ways to introduce parallelism in query processing to enable distributed stream processing.

2.2 Query Parallelism & Data Partitioning

There are two broad approaches of introducing parallelism in query processing (Valduriez, 2020):

1. **Inter-query parallelism.** This parallelism basically enables to run different queries in parallel. Although useful if there are many queries, a not very common case in data streaming, it doesn't help to process large volume streams, that is, streams with a high rate of tuples.

2. **Intra-query parallelism.** This parallelism actually enables to accelerate even a single query. It basically consists in parallelism the processing within the query. There are two mechanisms that can be used:

    a. **Inter-operator parallelism.** This parallelism lies in having multiple operators within the query plan to run in parallel and process tuples at the same time. It can be helpful if there are many operators, that again, it is not the common case in data streaming systems.

    b. **Intra-operator parallelism.** This type of parallelism parallelizes the processing of a single operator within the query plan, enabling multiple threads/processes to process different subsets of the incoming data streaming to the operator.

Centralized approaches already provide inter-query parallelism. One can run multiple queries on the same stream engine, and one could scale a little bit by using multiple centralized engines, one for each different query. Intra-query parallelism came with distributed data streaming frameworks. Some of the pioneering ones are Borealis (Daniel J. Abadi, 2005) and StreamCloud (V. Gulisano, 2010). Distributed data streaming systems introduced intra-query parallelism. Borealis introduced inter-query parallelism via inter-operator parallelism. While StreamCloud introduced intra-operator parallelism, thus enabling scale out to large volume data streams. This intra-operator parallelism pioneered by StreamCloud has become the standard in modern data streaming systems such as Flink (P. Carbone, 2015) (originally called stratosphere (A. Alexandrov, 2014)) and Spark streaming (Matei Zaharia, 2013). StreamCloud was the main outcome of the Stream FP7 project and has become one of the main references in distributed data streaming. It was also the first data streaming system to implement elasticity (V. Gulisano R. J.-M., 2012). This evolution of data streaming engines is depicted in **Error! Reference source not found.**.

2.3 Data Partitioning

Another aspect related to the query parallelism and needed to make such systems work is data partitioning. Once there is intra-operator parallelism, the question is how to partition the data across the different instances of a given operator. There are two big approaches for partitioning the data, vertical and horizontal. Vertical partitioning lies in splitting the data as different subsets of columns. However, data streaming queries are not very amenable for vertical partitioning in general. Horizontal partitioning splits data as disjoint sets of full rows/events. Horizontal partitioning can be further classified into range, hash partitioning, and round-robin. Range partitioning relies on each partition being a range of the distribution key. With hash partitioning the distribution key is hashed and the modulus is obtained dividing by a number of buckets. The result is a bucket number that is used as the distribution unit. In round-robin events are sent in a round-robin fashion to the parallel operators.

However, it should be noted that data partitioning depends on which category of operator is being applied. There are stateless and stateful operators. Stateless operators perform their function independently of any previous input. An example of stateless operators is a filter. Stateful operators on the other hand perform a computation over several rows/events. An example would be an aggregation.

Range partitioning has the disadvantage that it requires tuning to guarantee a good load balancing across parallel instances of the operator. Hashing on the other hand provides good load balancing by default, although it has an extra processing cost to compute the hash over the distribution key. Round-robin partitioning only works for stateless operators. If the partitioning is not done right, the parallelism does not improve the performance, simply wasting hardware.

2.4 Genuine Stream Processing vs. Batch Processing

Another dimension in the comparison of streaming engines is whether they are genuine stream processing engines or they process batches. Genuine stream processing engines, such as StreamCloud or Flink, process events as they are produced yielding real-time processing. However, batch-based engines such as Spark streaming have some delay in the processing of events (e.g a few seconds), since they aim to buffer a set of events before processing begins. Processing in batches can be advantageous for more efficient processing. However, it introduces a delay of seconds that can be detrimental for event-oriented applications.

2.5 Fault Tolerance and Message Processing Coherence Guarantees

Another aspect of distributed stream engines is whether they provide a mechanism to attain high availability in the advent of failures or not, and in affirmative case what are the message processing coherence guarantees.

There are different fault tolerance techniques that can be used in data streaming to provide high availability of different degrees:

Active Replication:

> It lies on having each operator instance running on two or more nodes and then all instances will receive the same data and produce the same output. This requires sending data from a replicated operator to another replicated operator and guarantee 1-copy semantics, that is, guaranteeing that the replicated data streaming engine has the same functional behaviour as the non-replicated one.

Checkpointing:

> Under a checkpointing approach, the state of stream processing pipeline components are periodically saved to a persistant storage medium. During operation, the stream processing platform tracks what parts of the stream have finished processing. If a failure occurs, then the pipeline (either as a whole or as only a subset of failed components) are rolled-back to the last checkpoint and processing continues from the start of that checkpoint.

2.6 Distributed Data Streaming Engine Components

The architecture of distributed data streaming engines shares some similarities from a functional point of view. They have, in general, the following layers:

1. Data ingestion layer.

   This layer is specialized to capture data from different data sources. These data sources can be monitoring probes or logs, information systems, IoT devices, etc. Basically, the different system specializes in being able to capture this data with minimal effort providing ways to automatically extract the data from the sources or supporting a data format that the data sources use such as text, JSON, binary, etc. They can also use a generic mechanism to connect to the data sources such as sockets or REST interfaces.

2. Data processing layer.

   This layer is in charge of actually processing the continuous queries. One of the most common ways is that the data processing layer is represented as a set of containers where one can deploy one or more data streaming operators. Data streaming operators are typically algebraic operators able to do basic functions such as filtering with a predicate, doing a vertical projection (i.e., selecting a subset of the columns), doing an aggregation (e.g., the sum of some column), or even joining two data streams (e.g., join together tuples with the same key).

3. Storage layer.

   Although data streaming engines are typically managing in memory data, many of them provide interfaces to storage of different kinds. It can be from file systems such as HDFS, to key-value data stores such as HBase or Cassandra or even full-fledged relational SQL databases such as PosgreSQL.

4. Output layer

   The output layer provides the interface to send the output of the continuous streaming queries running on the streaming engine. The output can be dashboards, visualization tools, a file, or even a socket to connect to an arbitrary application.

5. Management layer

   This is the layer handling the deployment and decommissioning of queries, fault-tolerance, etc. It orchestrates the different nodes used for the distributed data streaming engine to process a set of continuous data streaming queries and connect them to the data sources and produce the output data to the data sinks.

2.7 Distributed Data Streaming Engine Categories

Distributed data streaming engines have undergone specialization and the following different categories can be identified:

1. General purpose data streaming frameworks.

   They aim at providing a framework where continuous data streaming queries can be deployed and processed at different scales. They allow to express queries either as an acyclic directed graph of algebraic operators or in a query language. Early data streaming engines, such as Borealis and StreamCloud, basically they allow to process queries, while modern frameworks such as Flink and Spark Streaming they provide other functionalities needed by enterprises. For instance, Spark Streaming is integrated with Spark for doing machine learning.

2. Complex Event Processing (CEP).

   They provide support to write business rules to deal with the identification of particular events from continuous streams of information, such as a threat situation in security, when to buy or sell stocks in stock trading, etc. These rules enable to detect event patterns, abstract events or event-driven processes, model event hierarchies, detecting event relationships (causality, membership, timing), and do similar processing as with data streaming systems such as filtering, aggregation, and transformation. Examples of CEP systems are Esper (Espertech, s.f.) and StreamBase (Tibco, s.f.).

3. Online machine learning systems.

   These systems apply machine learning over data streams to provide a continuous learning over the data stream. This is interesting when one cannot train over a full dataset or new patterns can appear over time. One sample system in this category is SAMOA (Scalable Advanced Massive Online Analysis) (Gianmarco De Francisci Morales, 2015). SAMOA actually can work over different data streaming engines such as Storm (http://storm.apache.org/, s.f.), S4 (http://incubator.apache.org/s4, s.f.), and Samza (http://samza.incubator.apache.org, s.f.).

4. Streaming graph analytics.

   They basically keep a graph in memory updated by means of streaming actions and provide real-time processing over the graph such as recommendations, etc. The examples on this area mainly come from Twitter such as GraphJet (A. Sharma, 2016).

2.8 Window Programming Models

Data streaming engines work on the idea of an infinite stream of events. It is equivalent to a regular database table with infinite rows. The processing is made in chunks. Actually, a sliding window over this stream of events. Windows can be defined based on time or number of events. The nature of the window sliding can be different and basically, there are three main window programming models:

1. Fixed or tumbling windows.

These windows split time in consecutive intervals. Events are considered in the interval their timestamp belongs.

2. Sliding windows.

   Sliding windows are more generic. They support overlapping windows. They are defined by two parameters: length of the window and slide. The length indicates how long is the interval. The slide how much is shifted the window during each step. If the length is 10 and the shift 5, there will be windows from 1 to 10, 5 to 15, 11 to 20, etc. If the shift is equal to the length, then they behave like fixed windows.

3. Session windows.

   Sessions are defined by certain thresholds, typically certain time of inactivity. They are used for user-oriented input in which users are active and then after they are inactive for some time the session is considered to be finished. When activity restarts it starts in a new session.

2.9 Data Source Interaction Models

There are basically two modes of interaction with data sources: push and pull. In the push mode, the data source sends data through an API as soon as it has new data. In the pull mode, there is an agent at the data source side that periodically checks whether there is data available and sends it when new data is found. The push mode gives the best response times because data is sent as soon as it is available. Also the pull mode has the shortcoming that the frequency of pulling has to be higher than the frequency of data generation, since otherwise it leads to data loss.

3 The Landscape of Data Pipelining at Enterprises Today

In the current data management landscape, there are clearly two big families of data management, streaming data and data at rest. The latter is the most extended, however, the former is gaining traction to solve problems not amenable for the latter. There are some key differences between data streaming and persistent data stores. The first difference is the fact that data streaming queries are continuous and work over sliding windows, while persistent databases perform point-in-time queries executed just once over stored data. The second difference is that data streaming engines rely on in memory state that allow them to process efficiently large volumes of streaming data while persistent databases have to access persisted data that is far more costly and is what makes them slower and not being able to process the same volume of streaming data per node.

However, each family has a number of possibilities, especially, the one related to persistent databases. Let us first have a look at this landscape to better understand how INFINITECH can help in the problem of data pipelining.

Persistent databases can be classified first into:

1. Operational databases.

   These databases store data in persistent media. They allow to update the data while the data is being read. The consistency guarantees that are given with concurrent reads and writes vary. Operational databases, because they can be used for mission critical applications, might provide capabilities for attaining high availability that tolerates node failures and in some cases they can even tolerate data centre disasters leading to the whole loss or lack of availability of a whole data centre. The source of these disasters can be from a natural disaster like a flood, a fire, the loss of electric power, the loss of Internet connectivity, a Distributed Denial of Service attack resulting in the loss of CPU power and/or network bandwidth, or the saturation of some critical resource like DNS, etc.

2. Data warehouses.

   Data warehouses are informational databases. They are designed only to query data after ingesting it. They do not allow modifications, simply loading the data, and after the loading is complete, querying the data. They specialize on speeding up the queries by means of OLAP (On Line Analytical Processing) capabilities. OLAP capabilities are attained by introducing intra-query parallelism typically in the form of intra-operator parallelism. They typically use a customised storage model to accelerate the analytical queries by using a columnar model or they use an in-memory architecture.

3. Data lakes.

   They are used as scalable cheap storage where to keep historical data at affordable prices. The motivation of keeping this historical data might be legal requirements of data retention but more recently the motivation is from the business side to have enough data to be able to train machine learning models in a more effective way by reaching a critical mass of data in terms of time, but also in terms of detail. Some organizations use data lakes as cheap data warehouses when the queries are not especially demanding in terms of efficiency. A data lake might require more than an order of magnitude higher resources for an analytical query with a target response time than a data warehouse, while the price follows an inverse relationship.

Operational databases can themselves be classified in three broad categories:

1. Traditional SQL databases.

   Traditional SQL operational databases are characterized by two facts. The first one is that they provide SQL as query language. The second one is that they provide the so-called ACID guarantees over the data. We discuss later these ACID properties in detail. The main limitations of traditional SQL databases is their scalability, typically they either don't scale out or scale out logarithmically that means that their cost grows exponentially with the scale of the workload to be processed. They typically provide mechanisms for high availability that guarantee the ACID properties what is technically known as 1-copy consistency guarantees. The second limitation that they have is that they ingest data very inefficiently so they are not able to insert or update data at high rates. Their lack of linear scalability also results in exponentially growth of cost.

2. No-SQL databases

   No-SQL databases is a category with a number of different kinds of databases that are characterized by addressing requirements not well-addressed by traditional SQL databases. There are four main kinds of No-SQL databases as we will see later. Basically, they address the lack of flexibility of the relational schema that is very rigid and forces to know in advance all the fields of each row in the database and they are very disruptive when this schema has to be changed, typically resulting in having the database or at least the involved tables not available during the schema change. No-SQL databases fail to provide ACID consistency guarantees. On the other hand, most of them they are able to scale out, although not all kinds have this ability. Some of them are able to scale out but not linearly or not to large numbers of nodes.

3. NewSQL databases

   NewSQL databases appear as a new approach to address the requirements of SQL databases but trying to remove part or all of their limitations. The direction of NewSQL databases lie in bringing new capabilities to old traditional SQL databases by leveraging approaches from NoSQL and/or new data warehouse technologies. Some try to improve the scalability of storage. That is normally achieved by relying on some NoSQL technology or adopting an approach similar to some NoSQL technology. Scaling queries was an already solved problem. However, scaling inserts and updates had two problems. The first one is the inefficiency of ingesting data. The second one is that inability to scale out to large scale the ACID properties, that is, transactional management. Others have tried to overcome the lack of scalability of the ingestion while others the lack of scalability of transactional management.

NoSQL databases have different flavours and typically are divided into four categories:

1. Key-value data stores.

   They are schema-less and allow any value associated to a key. In most cases they attain linear scalability. Basically, each instance processes a fraction of the load. Since operations are based on an individual key-value pair, the scalability does not pose any challenge and most of the times is achieved. The schema-less approach provides a lot of flexibility. Basically, each row can have a totally different schema. Obviously, that is not how they are used. But they allow to evolve the schema without any major disruption. Of course, the queries have to do the extra work of being to understand rows with different schema versions, but since normally, the schema are additive, they add new columns or new variants, it is easy to handle. Key-value data stores excel at ingesting data very efficiently. Due to the fact that they are schema-less they can just store the data. This is very inefficient for querying, and normally they provide very little capabilities for querying such as getting the value associated to a key. In most cases they are based on hashing so they are unable to perform basic range scans. Example of key-value data stores are Cassandra and DynamoDB.

2. Document oriented databases

   They support semi structured data normally written in a language such as JSON or XML. Their main capability is that being able to store data in one of these languages efficiently and being able to perform queries for these data in an effective way. Representing these data in SQL is just a nightmare and doing queries of this relational schema even a worse nightmare. That is why they have succeeded. Some of them scale out in a limited way and not linearly, whilst some others do better and scale out linearly. The main shortcoming is that they do not support the ACID properties and that they are inefficient querying data that is structured in nature. Structured data can be queried one to two orders of magnitude more efficiently with SQL databases. Examples in this category are MongoDB and Couchbase.

3. Graph databases

   They specialize on storing and querying graph data. Graph data represented in a relational format becomes very expensive to query. The reason is that to traverse a path from a given vertex in the graph, one has to perform many queries, one per edge stemming from the vertex and as many times as the longest path sought in the graph. These results into too many client-server invocations. If the graph does not fit into memory, then it is even a bigger disaster since disk accesses will be involved for most of the queries. Also, the queries cannot be programmed in SQL and has to be performed programmatically. Graph databases on the other hand, they have a query language in which with a single invocation solve the problem. Data is stored to maximize locality of a vertex with contiguous vertexes. However, graph databases when they don't fit in a single node, then they start suffering from the same problem when they become distributed losing their efficiency and having a performance gain that is lost very soon with the system size in number of nodes. At some point a relational schema solution becomes more efficient than the graph solution for a large number of nodes. A widely used graph database is Neo4J.

4. Wide column data stores

   These data stores have more capabilities than key-value data stores. They typically perform range partitioning thus, supporting range queries. In fact, they might support some limited basic filtering. They are still schemaless. They also support vertical partitioning that can be convenient when the number of columns is very high. They have some notion of schema but still they are quite flexible in it. Example of this kind of data stores are BigTable and HBase.

In addition to the above categories, we have stream data managers already explored in the previous section and systems like Kafka that are streaming data managers but are persistent, and machine learning infrastructure such as Map Reduce, Spark and Pandas. Large organizations such as the ones in the Finance and Insurance verticals, typically have databases of many of the above kinds, with many instances of each category using the same brand or even different brands.

The main issue is that for analytical pipelines they have to move data across databases, many times having to adapt, modify or enrich the data. For this, ETL (Extract Transform Load) tools have been being used to perform batch processing and moving data from one database into another transforming the data as necessary. More recently a different approach such as ELT (Extract Load Transform) has been used. However, batch processing is not always feasible and it is required to acquire the data in real-time or near real-time when it is updated. For this purpose, CDC (Change Data Capture) infrastructure has been created that enables to get the changes from an operational database and then do something with these changes like storing it in some other database or do some processing like triggering events. In this task, we envision to take benefit from the CDC infrastructure in order to create the Intelligent Data Pipeline that INFINITECH needs to provide.

4 Intelligent Data Pipeline: The INFINITECH Approach

In INFINITECH, we are looking at how to simplify these data pipelines and adopt a uniform simple approach for them. Data pipelines get complicated mainly due to the mismatch of capabilities across the different kinds of systems. Many times data pipelines get very complex because of real-time requirements. One solultion is the adaptaion of an architecture, which is well known as **lambda architecture**.

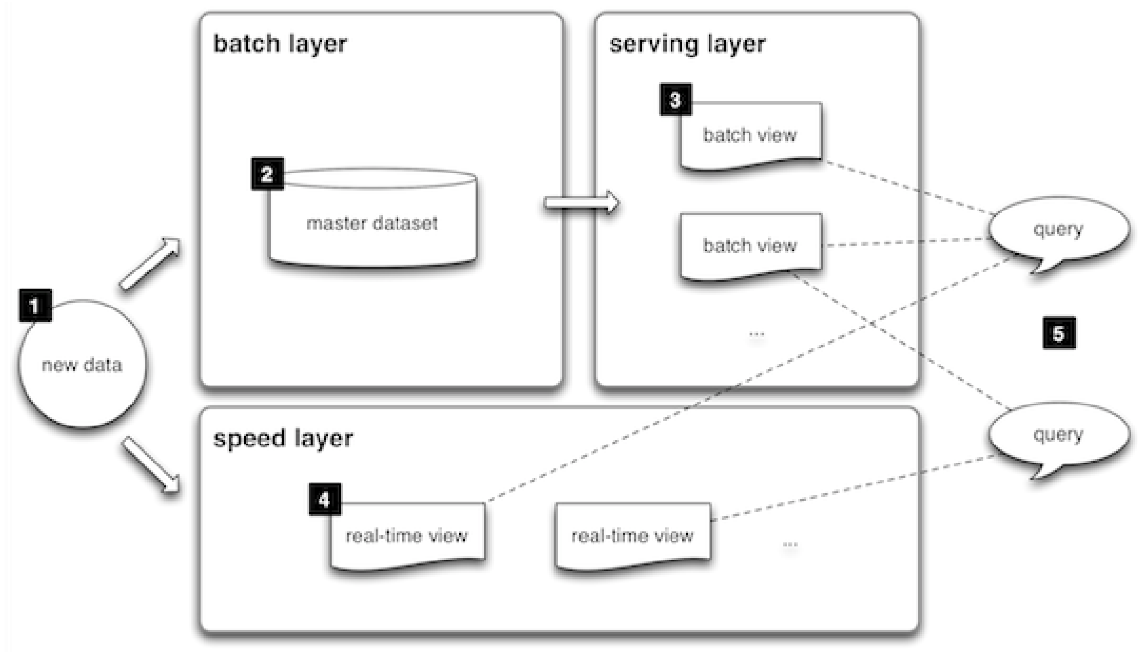

Figure 2: A typical lambda architecture

The lambda architecture combines techniques from batch processing with data streaming to be able to process data in a real-time manner. The lambda architecture is motivated by the lack of scalability of operational SQL databases. The architecture consists of three layers:

1. Batch layer.

   It is based on append only storage, typically a data lake, such as HDFS. Then, it relies on map-reduce for processing new batches of data in the forms of files. This batch layer provides a view in a read-only database. Depending on the problem being solved, the output might need to fully re-compute all the data to be accurate. After each iteration, a new view of the current data is provided. This approach is quite inefficient but it is solving a scalability problem that when it was invented did not have a good solution, the processing of tweets in Twitter.

2. Speed layer.

   This layer is based on data streaming. In the original system at twitter it was accomplished by the Storm data streaming engine. It basically processes new data to complement the batch view with the most recent data. This layer does not aim for accuracy, which is usually very crucial for applications in the insurance and finance sector, but it provides more recent data to the global view achieved with the architecture.

3. Serving layer.

The serving layer processes the queries over the views provided by both the batch and speed layer. Batch views are indexed to be able to answer queries with low response times and combines them with the real-time view to provide the answer to the query combining both real-time data and historical data. This layer typically uses some key-value data store to implement the indexes over the batch views.

The main shortcoming of the lambda architectures is its complexity and the need to have totally different code bases for each layer that have to be coordinated to be fully in sync. Maintenance of the platform is very hard since debugging implies understanding the different layers with totally different natures, involved technologies and approaches.

Other more traditional architectures are based on combining an operational database with a data warehouse. The operational database deals with more recent data while the data warehouse deals with historical data. In this architecture, queries can only see either the recent data or historical data, but not a combination of both as it was done in the lambda architecture. In this architecture there is a periodic process that copies data from the operational database into the data warehouse. This periodic process has to be performed very carefully since it can hamper the quality of service of the operational database. This periodic process is most of the time achieved by ETL tools. Many times this process is performed over the weekends in businesses where their main workload comes during weekdays. Another problem that this architecture exhibits is the fact that the data warehouse typically cannot be queried while it is being loaded, at least the tables that are being loaded. This forces to split the time of the data warehouse into loading and processing. When the loading process is daily, finally the day is split into loading and processing. The processing time consumes a fraction of hours of the day that depends on the analytical queries that have to be answered daily. It leaves a window of time for loading data that is the remaining hours of the day. At some point data warehouses cannot ingest more data because the loading window is exhausted. We name this architecture **current-historical data splitting**.

Due to the saturation of the data warehouse is a common problem, another architectural pattern has been devised to deal with this issue in an architectural pattern that we name **data warehouse offloading**. This pattern relies in creating small views of the data contained by the data warehouse and store them on independent databases, typically called **data marts**. Depending on the size of the data and the complexity of the queries data marts can be handled by operational SQL databases or they might need a data manager with OLAP capabilities that might be another data warehouse or a data lake plus an OLAP engine that works over data lakes.

In some other cases, the problem lies in the fact that the operational database cannot handle the whole workload due to its lack of scalability and part of this workload can be performed without being real-time. In these cases, a copy of the database or the relevant part of the data of the database is copied into another operational database during the time that the operator is not being used, normally, weekends or nights, depending on how long the copy of the database takes. If it takes less than the night time is performed daily. If it takes more than that time is performed over the weekend. If it takes more than then weekend, it cannot be done with this architectural pattern. We call this architectural pattern **database snapshotting.**

In other cases, there are real-time or quasi real-time requirements and the database snapshotting does not solve the problem. In this case, a CDC (Change Data Capture) system is used that captures changes in the operational data and inject them into another operational database. The CDC is only applied over the fraction of the data that will be processed by the other operational database. The workload is not performed over the operational database due to technical or financial reasons. The technical reason is that the operational database cannot handle the full workload and some processes need to be offloaded to another database. The financial reason is that the operational database can handle the workload but the price is too high. The latter typically happens with the mainframe. We name this architecture **operational database offloading**.

A very typical and important workload that lies in having to ingest high volumes of detailed data and compute recurrent aggregate analytical queries over this detailed data. This workload has been addressed with more specific architectures. One of such architectures uses an operational database for ingesting the detailed data, and uses another operational database to store aggregated views of the data. These aggregated views are generated periodically by means of an ETL process that traverses the data from the previous period in the detail operational database, computes the aggregations and store them in the aggregate operational database. The recurrent queries are processed over the aggregation database. Since the database contains already the pre-computed aggregates the queries are light enough to be computed at an operational database. We call this architecture **detail-aggregate view splitting**. One of the main shortcomings of this architecture is the fact that the aggregate queries have an obsolete view of the data since they miss the data from the last period. Typical period lengths go from 15 minutes to hours or a full day. Some times this architecture is solved.

The kind of operational databases typically used for the above architecture are SQL operational databases and since they do not scale, they require to use an additional architectural pattern that we call **database sharding**. Sharding lies in overcoming the lack of scalability or linear scalability of an operational database by storing fractions of the data on different database independent servers. Thus, each database server handles a workload small enough, and by aggregating the power of many different database manager instances the system can scale. The main shortcomings of this architecture lie in that now queries cannot be performed over the logical database, since each database manager instance only knows about the fraction of data it is storing and cannot query any other data. Another major shortcoming lies in that there are no transactions across database instances meaning that is stored data across different instances are related, they don't have consistency guarantees neither in the advent of concurrent reads and writes or in the advent of failures.

Other systems tackle the previous problem of recurrent aggregate queries by computing the aggregates on the application side using the memory. So basically, this in-memory aggregates are computed and being maintained as time progresses. The recurrent aggregation queries are solved by reading this in-memory aggregations, while access to the detail data are solved by reading from the operational database, many times using sharding. We name this architectural pattern **in-memory application aggregations**.

Another approach to deal with recurrent aggregate queries at scale lies in what we call **federated aggregations**. The architectural pattern lies in using sharding to store fractions of the detail data and then use a federator at application level that basically queries the individual sharded database managers getting the resultsets of the individual aggregate queries and then, aggregate them manually to compute the aggregate query over the logical database. This architectural pattern is applied frequently for monitoring systems and it is called in that context Monitor of Monitors (MoM).

In INFINITECH we envision a holistic solution to the issue of data pipelining that works with all kinds of storage, handling efficiently aggregates, and addresses the need for temporal storages to deal with snapshot databases. This holistic solution aims at minimizing the number of storage systems needed to develop an analytical pipeline and addressing all of the above identified architectural patterns for data pipelining. It also aims at unifying the data pipelines combining data streaming and data at rest.

In what follows we provide the list of targeted architectural patterns for data pipelining and how they will be automated and solved with INFINITECH's innovations in the data management layer, which are incorporated into the INFINISTORE data store

1. **Lambda architecture.**

   In INFINITECH the lambda architecture is totally trivialized by removing the at least three data management technologies and three different code bases with ad hoc code for each of the queries and just having a single database manager with declarative queries in SQL. The lambda architecture is simply substituted by the INFINISTORE, which relies on the LeanXcale database. LeanXcale scales out linearly its operational storage solving one of the key shortcomings of operational databases that motivate the lambda architecture. The second obstacle from operational databases was its inefficiency in ingesting data that makes them too expensive even for data ingestions they can manage. As the database grows, the cache is rendered ineffective and each insert requires to read a leaf node that requires first to evict a node from the cache and write to disk. This means that every write requires two IOs. LeanXcale solves this issue by providing the efficiency of key-value data stores in ingesting data thanks to the blending of SQL and NoSQL capabilities due to the use of a new variant of LSM trees. With this approach, updates and inserts are cached in an in-memory search tree and periodically propagated all together to the persisted B+ tree. Thanks to this approach the locality of updates and inserts on each leaf of the B+ tree is greatly increased amortizing the cost of each IO among many rows. The third issue solved by INFINISTORE that is not solved by the lambda architecture, it is the ease to query. The lambda architecture requires developing programmatically each query with three different code passes for each of the three layers. Using the INFINITECH data management layer and its INFINISTORE, queries are written in simple and widely known SQL. SQL queries are automatically optimized unlike the programmatic queries in the lambda architecture that require manual optimization across three different code basis for each of the layers. The fourth issue that is solved is the one of the cost of recurrent aggregation queries. In the lambda architecture, this issue is typically solved in the speed layer using data streaming. In INFINISTORE, with the development and adaption of the online aggregates, we enable real-time aggregation without the problems of operational databases and providing a low cost solution with low response time.

2. **Current-Historical Data Splitting.**

   In this approach, data is split between an operational database and a data warehouse or a data lake. The current data is kept on the operational database and historic data in the data warehouse or data lake. However, queries across all the data are not supported with this architectural pattern. In INFINITECH a new pattern will be used to solve this problem named **Real-Time Data warehousing.** This pattern will be solved by a new innovation that will be introduced in LeanXcale, namely, the ability to split analytical queries over LeanXcale and an external data warehouse. Basically, it will copy older fragments of data into the data warehouse periodically. LeanXcale will keep the recent data and some of the more recent historical data. The data warehouse will keep only historical data. Queries over recent data will be solved by LeanXcale, and queries over historical data will be solved by the data warehouse. Queries across both kinds of data will be solved in the following way. If they do not contain joins, basically the query will be executed with a predicate over time on both databases guaranteeing a split without overlapping and without missing any data item and the union of both results will be returned as the result of the query. If the query contains joins then it will be split into four subqueries. One subquery doing the joins across recent data that will be pushed down to LeanXcale. One subquery doing joins across historical data that will be pushed down to the data warehouse. And a third subquery doing joins across recent and historical data that will be solved at LeanXcale that will used the data warehouse as external data source for reading the data. In this way, the bulk of the historical data query is performed by the data warehouse, while the rest of the query is performed by LeanXcale. This approach enables to deliver real-time queries over both recent and historical data giving a 360 degree view of the data.

3. **Data warehouse offloading.**

In data warehouse offloading due to the saturation of the data warehouse data marts are used using other database managers and making a more complex architecture that requires multiple ETLs and copies of the data. Within INFINITECH this issue can be solved in two ways: One way is by using operational database offloading to INFINISTORE with the dataset of the data mart. The advantage of this approach with respect to data warehouse offloading is that the data mart contains data that is real-time, instead of obsolete data copied via a periodic ETL. The second way is to use database snapshotting taking advantage of the fast speed and high efficiency of loading of LeanXcale. In this way, a data mart can be created periodically with the same or higher freshness than a data mart would have. The advantage is that the copy would come directly from the operational database instead of coming from the data warehouse thus resulting in fresher data.

4. **Database snapshotting.**

In INFINITECH database snapshotting can be actually be avoided by using its data management layer as the operational database. This can be done thanks to the linear scalability of the INFINISTORE that does not require offloading part of the workload to other databases. However, in many cases, organizations are not ready to migrate their operational database because of the large amount of code relying on specific features of the underlying database. This is the case with mainframes with large Cobol programs and batch programs in JCL. In that case, INFINITECH by relying on LeanXcale can provide a more effective snapshotting or even able to substitute snapshotting by operational database offloading that provides full real-time data. In the case of snapshotting, thanks to the efficiency and speed of data ingestion of LeanXcale, snapshotting can be performed daily instead of weekly since load processes that takes days are reduced to minutes. But snapshotting can be substituted by operational database offloading thanks to the scalability and speed of ingestion of LeanXcale. The main benefit is that data freshness changes from weekly to real-time. This speed in ingestion is achieved thanks to LeanXcale capability of ingesting and querying data with the same efficiency independently of the dataset size. This is achieved by means of bidimensional partitioning. The bidimensional partitioning exploits the timestamp in the key of historical data to partition tables on a second dimension. Tables in LeanXcale are partitioned horizontally through the primary key. But then, they are automatically split on the time dimension (or an auto-increment key, whatever is available) to guarantee that the table partition fits in memory and thus, the load becomes CPU bound and thus, very fast. Traditional SQL databases get slower as data grows due to the B+ tree used to store data becomes bigger in both terms of number of levels and number of nodes. LeanXcale thanks to bi-dimensional partitioning keeps the time to ingest data constant. Queries are also speeded up thanks to intra-operator parallelization of all algebraic operators below joins.

5. **Operational database offloading.**

One of the main limitations of the operational database offloading is the fraction of data offloaded to a single database. Typically, this approach is performed with mainframes that can process very high workloads that soon overload other operational databases with much more limited capacity and incapable of scaling. Again by relying on LeanXcale, INFINITECH will overcome these limitations. LeanXcale can even support to full set of changes performed over the mainframe thanks to its scalability so it does not set any limitation on the dataset size and rate of data updates/inserts over this dataset.

6. **Detail-aggregate view splitting.**

   In INFINITECH this pattern is totally removed because it is not needed anymore. By taking advantage of its declarative real time analytical framework and the so called *online aggregates* developed under the scope of T5.3, aggregate tables are built incrementally as base data is inserted. This implies to increase the cost of ingestion, but since LeanXcale is more than one order of magnitude more efficient than the market leader, it means that it can still ingest the data more efficiently despite the online aggregation, but then, recurrent aggregation analytical queries become costless since they only have to read a single row or a bunch of rows to provide the answer thanks to the fact that each aggregation has been already computed incrementally.

7. **Database sharding.**

   Database sharding is not needed in INFINITECH thanks to the linear scalability of its INFINISTORE. Thus, what before required programmatically splitting the data ingestion and data queries across independent database instances, now, it is not needed anymore. LeanXcale is able to scale out linearly to hundreds of nodes.

8. **In-memory application aggregations.**

   In INFINITECH in-memory application aggregations are not needed anymore removing all the problems around them like the loss of data in the advent of failures and what is more all the development and maintenance cost of the code required to perform the in-memory aggregations. Not only that in-memory aggregations work as far they can be computed in a single node, when multiple nodes are required they become extremely complex and in most cases out of reach of technical teams. In INFINITECH the online aggregates from LeanXcale will be leveraged to compute the aggregations for recurrent aggregation queries. LeanXcale keeps internally the relationship between tables (called parent tables) and aggregate tables built from the inserts in these tables (called child aggregate tables). When aggregation queries are issued, the query optimizer has been enriched with new rules to automatically detect which aggregations on the parent table can be accelerated by using the aggregations in the child aggregate table. This results in transparent improvement of all aggregations in the parent table by simply declaring a child aggregate table (obviously of the ones that can exploit the child table aggregates). More information can be found at the relevant D5.4 and D5.5 deliverables ("Framework for Declarative and Configurable Analytics")/

9. **Federated aggregations.**

   Federated aggregations share the motivation of in-memory aggregations but basically enable them to extend to a multiple set of nodes. As with in-memory application aggregations, INFINITECH fully solve the problem in a trial way by relying on its online aggregates.

10. **Streaming data and data at rest.**

Several applications require to combine streaming data with data at rest. In INFINITECH these applications will be addressed by using different mechanisms. When streaming data needs to be correlated with persistent data it will be attained by means of the integration of INFINISTORE with Flink. When streaming data needs to produce a persistent output, it will also be addressed by means of the Flink and INFINISTORE integration. However, sometimes this integration results complex because it implies writing queries in two different subsystems and it is complex their integration. For this reason, we will develop an integration of SQL with an SQL-like query language for streaming data in the form of a query language that integrates both the access to data at rest and the access to streaming data. By using a unified language, it becomes trivial the use of a column or set of columns from a streaming tuple in a query performed over the persistent storage and similarly, to integrate the output of an SQL query into the output stream of a data streaming operator that correlates streaming data with persistent data.

11. **NoSQL and SQL data.**

    As previously discussed organizations have a myriad of different kinds of database managers that include both SQL and NoSQL databases. The main issue is that data stored on each family of data stores belongs to a single logical database of the organization and this split is artificial due to the technical limitations of different database technologies that prevent from using a single database for all kinds of data. In INFINITECH polyglot support will be provided to solve the data pipelines across different families of databases. Polyglot data support will enable to query from a common endpoint to SQL data stored in LeanXcale or other SQL databases and data stored in key-value data stores, wide-column data stores, document-oriented data stores, and graph databases.

Finally, INFINITECH will also integrate the different tools require to support all the data pipelines including Change Data Capture (CDC) systems and ETL tools to provide a holistic solution to the automation of data pipelining.

5 Intelligent Data Pipeline in practice

In this section we provide insights along with a concrete example on how to make use of the INFINITECH Intelligent Data Pipeline, instead of using complex architecture designs as previously presented. We demonstrate how to setup, configure and deploy such a data pipeline in order to move data from an operational datastore to INFINISTORE. We envision the scenario that an operational datastore is accepting the transactional workload, however, it cannot be used to efficiently execute analytical queries and thus, data (or fragments of data) should be migrated either to a data warehouse or to another operational datastore. Typically, these use cases are solved by applying architecture designs such as **operational database offloading** or **current-historical data splitting.** Even if our scenario is limited to these architectures, our solution is generic and can be used to implement each type of data pipeline, as the datastore to migrate data to or from, is the INFINISTORE, which can be used to solve all the architectures in a holistic way.

In our example, we make use of MySQL datastore that will take the role of the operational database of a finance or insurance enterprise. MySQL is widely used and ensures database transactions and provides relational query processing capabilities. However, it suffers when it comes to analytics under operational workload. Due to this, there is often the need to migrate periodically data to a data warehouse that can be used instead for such type of processing. The data migration often takes place as a batch process that takes place during night periods that the database is almost idle and takes a lot of time, which can typically can be couple of hours. If the process fails during the night, the data warehouse is not updated, as there is no time to repeat such a time-consuming process, resulting in lack of data coming from the last day. With the INFINITECH data pipeline, we want to send data from the operational datastore to the INFINISTORE when a data modification operation takes place. This way, the INFINISTORE will always have fresh data, without having to wait for the night when the batch processing takes place. Moreover, as INFINISTORE provides Hybrid Transactional and Analytical Processing (HTAP) capabilities, we can offload data to the latter in real-time, no matter the rate of data ingestion, while at the same time, it can be used to for advanced analytical processing. This is due to the outcomes of the task T3.1 “Framework for Seamless Data Management and HTAP”, with the reader being advised to go through the corresponding report D3.2 “Hybrid Transactional/Analytics Processing for Finance and Insurance Applications – II” for more details.

For the implementation of the INFINITECH Intelligent Pipelines, we rely on the Debezium[3] that implements the Change Data Capture (CDC) paradigm. Before getting into more details, the following subsection provides a general overview of what Debezium can offer.

5.1 Use of Debezium for Change Data Capture

Debezium is an open source distributed platform that can be used for implementation that relies on the change data capture. It provides a set of distributed services that can monitor changes in a database schema, capture those changes in a per data-item level, whether these changes are related with an insertion, update or delete of a data record, and publishes these changes so that other components can react. Other components might be both software and application level components, other datastores and steaming processing frameworks. In INFINITECH, we make use of it to i) send data modifications into the INFINISTORE and ii) send data modifications of INFINISTORE to other data processing frameworks. The latter will be the case of the Semantic Interoperability engine that is being built under the scope of WP4.

[3] https://debezium.io/

What Debezium does is that it keeps a transaction log that stores data modification operations that happens in a per data item level in the *source* database, when data is finally committed, and propagates this log to different listeners that can react either by triggering specific events, or store the data in a persistent storage medium of the *target* database that listens to this log. That way, it implements the *change data capture* paradigm which allows the user to monitor and capture data changes that takes place in the *source* and send these changes to the *target*. Typically, the *source* might be an operational datastore and the *target* a data warehouse. As a result, this approach clearly fits to the scenario that we demonstrate.

According to its official web site, Debezium provides support for data connectivity for a variety of different database vendors, such as *MySQL database servers, MongoDB replica sets or sharded clusters, PostgreSQL servers and SQL Server databases.* This allows us to rely on this framework to migrate data from different vendors that are dominant in the insurance and finance sector. Moreover, it provides an interface that can be used by system developers to create additional connectors to other mediums. In INFINITECH, we have developed a connector to allow the interaction with INFINISTORE.

A typical deployment of Debezium consists of various components that are involved in the data pipeline for the change data capture. Typically, a cluster of Kafka brokers as the medium for Debezium to exchange the transaction logs among the involved listeners. The Debezium records all the events of data modification and stores them as transaction logs in Kafka, from which the application level components or other data management system can consume those logs and respond accordingly. Additionally, the deployment requires a Debezium connector that monitors the source database. We can have one connector per monitoring database. As connector capturers the data changes that happen in the source database, they persist the logs into Kafka, and from them, we can retrieve those logs and store the changes into INFINISTORE.

5.2 From an operational datastore to INFINISTORE

One important requirement found in many organizations in finance and insurance sector is to have an instance of an analytical database management system running in parallel with their databases while not changing their existing systems. The solution that INFINITECH proposes is to make use of its Intelligent Data Pipeline which achieves this is by the implementation of the change data capture, using Debezium. In this subsection, we explain how to integrate a MySQL database with INFINISTORE, although this approach is generic enough and can be used with other source databases, such as *PostgreSQL* or MongoDB.

Following the basic concepts of Debezium that were previously described, the overall architecture is depicted in Figure 3.

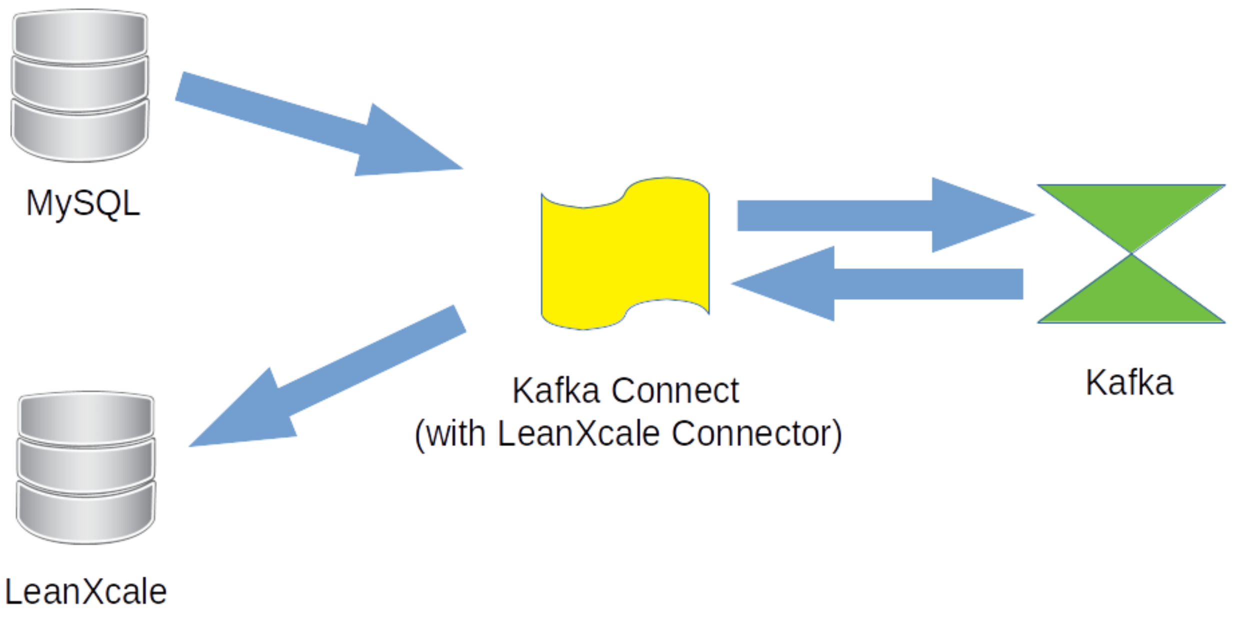

Figure 3: INFINITECH Data Pipeline moving data from MySQL to INFINISTORE

The overall architecture involves the two databases that need to exchange data modifications via the *change data capture*: the operational datastore of MySQL and the hybrid transactional and analytical database of INFINISTORE, which is based on the technology that LeanXcale is implementing under the scope of the project. Debezium is the medium to monitor changes in the MySQL side and sends transaction logs using a specific connector to Kafka. Kafka is used as the intermediate data queue that persistently stored data sent by the Debezium connector and are retrieved to be stored in LeanXcale. This also exploits the outcomes of task T3.1 “Framework for Seamless Data Management and HTAP”, and more precisely the implementation of the Kafka connector developed there and is being documented in the corresponding report D3.2 “Hybrid Transactional/Analytics Processing for Finance and Insurance Applications – II” for more details.

The creation of the components is performed with docker images that include, according to the architecture of Debezium, the following:

- Apache Zookeeper.
- Apache Kafka.
- Kafka Connect / Debezium image with the modification of the Kafka Connector for the INFINISTORE placed into the corresponding connect directory.
- An empty MySQL database image into which we perform some create statements.
- An empty instance of the INFINISTORE into which all changes from MySQL are replicated.

At this phase of the project, we have not migrated yet our solution to be compliant with the INFINITECH way of deployment, as the overall implementation is under validation. Therefore, we make use of the *docker-compose* to configure the deployment and integration of all the involved components. The docker-compose.yaml that someone can use is the following:

```
version: '2'
services:
  zookeeper:
    image: debezium/zookeeper:1.1
    ports:
     - 2181:2181
     - 2888:2888
     - 3888:3888
  kafka:
    image: harbor.infinitech-h2020.eu/interface/lx-kafka:latest
    ports:
     - 9092:9092
    links:
     - zookeeper
    environment:
     - ZOOKEEPER_CONNECT=zookeeper:2181
  mysql:
    image: debezium/example-mysql:1.1
    ports:
     - 3306:3306
    environment:
     - MYSQL_ROOT_PASSWORD=debezium
     - MYSQL_USER=mysqluser
     - MYSQL_PASSWORD=mysqlpw
  connect:
    image: debezium/connect:1.1
    ports:
     - 8083:8083
    links:
     - kafka
     - mysql
    environment:
     - BOOTSTRAP_SERVERS=kafka:9092
     - GROUP_ID=1
     - CONFIG_STORAGE_TOPIC=my_connect_configs
     - OFFSET_STORAGE_TOPIC=my_connect_offsets
     - STATUS_STORAGE_TOPIC=my_connect_statuses
```

It is noticed that we make use of a zookeeper instance that is needed by the Debezium, the MySQL database that will be the operational database from which will send data modifications to the INFINISTORE, the corresponding connector of Debezium that will monitor the MySQL to write the transaction logs and finally, the Kafka queue. The reader should also notice that we make use of the Kafka image that is provided by the INFINISTORE.

In order to configure the Debezium connector to monitor the MySQL database and send results to INFINISTORE, we need to provide the following configuration file:

```
{
    "name": "inventory-connector",
    "config": {
        "connector.class": "io.debezium.connector.mysql.MySqlConnector",
        "tasks.max": "1",
        "database.hostname": "mysql",
        "database.port": "3306",
        "database.user": "debezium",
        "database.password": "dbz",
        "database.server.id": "184054",
        "database.server.name": "dbserver1",
        "database.whitelist": "inventory",
        "database.history.kafka.bootstrap.servers": "kafka:9092",
        "database.history.kafka.topic": "schema-changes.inventory",
    "transforms": "route,unwrap",
        "transforms.route.type": "org.apache.kafka.connect.transforms.RegexRouter",
        "transforms.route.regex": "([^.]+)\\.([^.]+)\\.([^.]+)",
        "transforms.route.replacement": "$3",
    "transforms.unwrap.type": "io.debezium.transforms.UnwrapFromEnvelope",
    "transforms.unwrap.drop.tombstones": "false",
    "transforms.unwrap.delete.handling.mode": "none"
    }
}
```

This configuration is detailed explained in the Debezium documentation and is out of the scope of this report. However, to provide a brief explanation, it defines a MySQL Connector which will be used, provides the hostname and port where the operational database is deployed, the username and password to connect, along with information regarding the Kafka queue that will be used to send the transaction logs. It is important to also mention that in the MySQL connector, we include two transformations of "transforms": "route,unwrap". With the transformation route, the connector puts the messages into the topic using the table name. With the second transformation, unwrap, the original message is changed to be compatible with other connectors, such as the connector provided for INFINISTORE. Let's save this configuration in file named *register-mysql.json.* The Debezium connector provides a REST API that can be used to configure it, so the following command can actually be used:

```
curl -i -X POST -H "Accept:application/json" -H  "Content-Type:application/json"
http://localhost:8083/connectors/ -d @register-mysql.json
```

Now we would need to also configure the connector of the INFINISTORE. In our scenario, we will make use of the following configuration:

```
{
     "name": "lx-connector",
     "config": {
         "connector.class": "com.leanxcale.connector.kafka.LXSinkConnector",
         "tasks.max": "1",
     "topics": "t1",
         "connection.properties": "lx://lx:9876/db@APP",
     "auto.create": "true",
     "delete.enabled": "true",
     "insert.mode": "upsert",
     "batch.size": "500",
     "connection.check.timeout": "20",
     "sink.connection.mode": "kivi",
     "sink.transactional": "false",
     "table.name.format": "t1",
     "pk.mode": "record_key",
     "pk.fields" : "id",
     "fields.whitelist": "field1,field2"
     }
}
```

In this example, we indicate only one table (t1) in the connector where the table is auto-created at the first insert. We can delete rows by setting "delete.enabled": "true".

Similarly, we save the configuration in a file named *register-lx.json* and we can now configure the connector of INFINISTORRE using the REST API of Debezium, with the following command:

```
curl -i -X POST -H "Accept:application/json" -H  "Content-Type:application/json"
http://localhost:8083/connectors/ -d @register-lx.json
```

After having properly configured the Debezium connectors, we can start creating records to the MySQL operational data store, and see how everything works in practice. We first need to connect to the container of MySQL and we will create a table, containing three fields, and add a data row. The following lines should be executed from the MySQL command line client:

```
CREATE TABLE t1(id int, field1 int, field2 varchar(10), PRIMARY KEY(id));
INSERT INTO t1 VALUES (1, 1, 'one');
```

Taking a look at the logs, we should observe something like the following:

```
connect_1    | 2021-07-09 13:39:34,676 INFO   ||  using percentage to target request of new batch
0.85   [com.leanxcale.txnmgmt.lxinfo.client.TSProvider]
connect_1    | 2021-07-09 13:39:34,678 WARN   ||  Socket using nagle true
[com.leanxcale.txnmgmt.lxinfo.client.LeanxcaleInfoClient]
connect_1    | 2021-07-09 13:39:34,680 INFO   ||  LeanXcaleInfoClient initialized on port 58514
[com.leanxcale.txnmgmt.lxinfo.client.LeanxcaleInfoClient]
connect_1    | 2021-07-09 13:39:34,680 INFO   ||  LeanXcaleInfoClient obtained LXIS instances for
HA: {}   [com.leanxcale.txnmgmt.lxinfo.client.LeanxcaleInfoClient]
connect_1    | kv.Conn[1]: *** no.auth = 1
connect_1    | kv.Conn[1]: *** no.crypt = 2
connect_1    | kv.Conn[1]: *** no.flushctlout = 1
connect_1    | kv.Conn[1]: *** no.npjit = 1
connect_1    | kv.Conn[1]: *** no.tplfrag = 1
connect_1    | kv.Conn[1]: *** no.xmbiocuts = 1
connect_1    | kv.Conn[1]: *** no.dstids = 1
connect_1    | kv.Conn[1]: *** test.resize = 1
connect_1    | 2021-07-09 13:39:34,720 INFO   ||  Table
com.leanxcale.connector.kafka.utils.metadata.TableId@4e29b9 is not registered for the connector.
Checking db...   [com.leanxcale.connector.kafka.sink.impl.LXWriterImpl]
connect_1    | 2021-07-09 13:39:34,722 INFO   ||  Table t1 not found. Creating it
[com.leanxcale.connector.kafka.sink.impl.LXWriterImpl]
connect_1    | 2021-07-09 13:39:34,740 INFO   ||  Registering table t1 in connector
[com.leanxcale.connector.kafka.sink.impl.LXWriterImpl]
connect_1    | 2021-07-09 13:39:34,755 INFO   ||  Closing sessionFactory
[com.leanxcale.kivi.session.SessionFactory]
connect_1    | 2021-07-09 13:39:34,756 INFO   ||  Closing socket 58514
[com.leanxcale.txnmgmt.lxinfo.client.LeanxcaleInfoClient]
connect_1    | 2021-07-09 13:39:34,756 INFO   ||  Disconnecting
[com.leanxcale.txnmgmt.lxinfo.client.LeanxcaleInfoClient]
connect_1    | 2021-07-09 13:39:34,756 INFO   ||  Remote disconnected
[com.leanxcale.txnmgmt.lxinfo.client.LXInfoClientPeriodic]
connect_1    | kv.Conn[1]: metaclientproc aborted by user
connect_1    | 2021-07-09 13:39:34,757 INFO   ||  Socket closed
[com.leanxcale.txnmgmt.lxinfo.client.LeanxcaleInfoClient]
connect_1    | 2021-07-09 13:39:34,757 INFO   ||  Closing socket 58514
[com.leanxcale.txnmgmt.lxinfo.client.LeanxcaleInfoClient]
connect_1    | 2021-07-09 13:39:34,758 INFO   ||  Disconnecting
[com.leanxcale.txnmgmt.lxinfo.client.LeanxcaleInfoClient]
connect_1    | 2021-07-09 13:39:36,496 INFO   ||  Tuples inserted: 0 commited in last 10000 ms
[com.leanxcale.connector.kafka.insert.impl.CommitLogger]
connect_1    | 2021-07-09 13:39:36,496 INFO   ||  Tuples deleted: 0 commited in last 10000 ms
[com.leanxcale.connector.kafka.insert.impl.CommitLogger]
connect_1    | 2021-07-09 13:39:36,496 INFO   ||  Tuples upserted: 1 commited in last 10000 ms
[com.leanxcale.connector.kafka.insert.impl.CommitLogger]
connect_1    | 2021-07-09 13:39:36,496 INFO   ||  Tuples updated: 0 commited in last 10000 ms
[com.leanxcale.connector.kafka.insert.impl.CommitLogger]
connect_1    | 2021-07-09 13:39:40,015 INFO   ||  WorkerSourceTask{id=inventory-connector-0}
Committing offsets   [org.apache.kafka.connect.runtime.WorkerSourceTask]
connect_1    | 2021-07-09 13:39:40,015 INFO   ||  WorkerSourceTask{id=inventory-connector-0}
flushing 0 outstanding messages for offset commit
[org.apache.kafka.connect.runtime.WorkerSourceTask]
connect_1    | 2021-07-09 13:39:40,022 INFO   ||  WorkerSourceTask{id=inventory-connector-0}
Finished commitOffsets successfully in 7 ms   [org.apache.kafka.connect.runtime.WorkerSourceTask]
connect_1    | 2021-07-09 13:39:46,470 INFO   ||  Received 0 records
[com.leanxcale.connector.kafka.sink.LXSinkTask]
```

```
connect_1    | 2021-07-09 13:39:46,471 INFO   ||  WorkerSinkTask{id=lx-connector-0} Committing
offsets asynchronously using sequence number 1: {t1-0=OffsetAndMetadata{offset=1, leaderEpoch=null,
metadata=''}}   [org.apache.kafka.connect.runtime.WorkerSinkTask]
```

This shows us that the connector to the INFINISTORE has been triggered when we added a row, and the line with bold tells us that 1 record has been upserted to INFINISTORE. If we open an SQL client to the latter, we can see that the row has been now added transparently, with *id* as the primary key, having both fields *field1, field2* as defied in the *register-lx.json* configuration file. We can now add records in whatever high rate is supported by the source operational datastore, and taking benefit of the INFINISTORE capabilities developed within the project, we can transparently use the *change data capture* to implement our Data Pipeline and migrate data to INFINISTORE, in real time.

5.3 Use of Debezium for Change Data Capture with Avro Serialization

In this subsection, we continue working with Debezium for *change data capture* and INFINISTORE, configured for a common scenario that a finance organization has a large operational database and requires to use **operational database offloading** or other architecture to migrate data from the operational store to another data base management system. We still make use of MySQL as the source database, but instead, we configure Avro as the message format instead of JSON. Being a more compact message, it can offer significantly improved performance.

This time, our overall architecture is changed and is depicted in Figure 4.

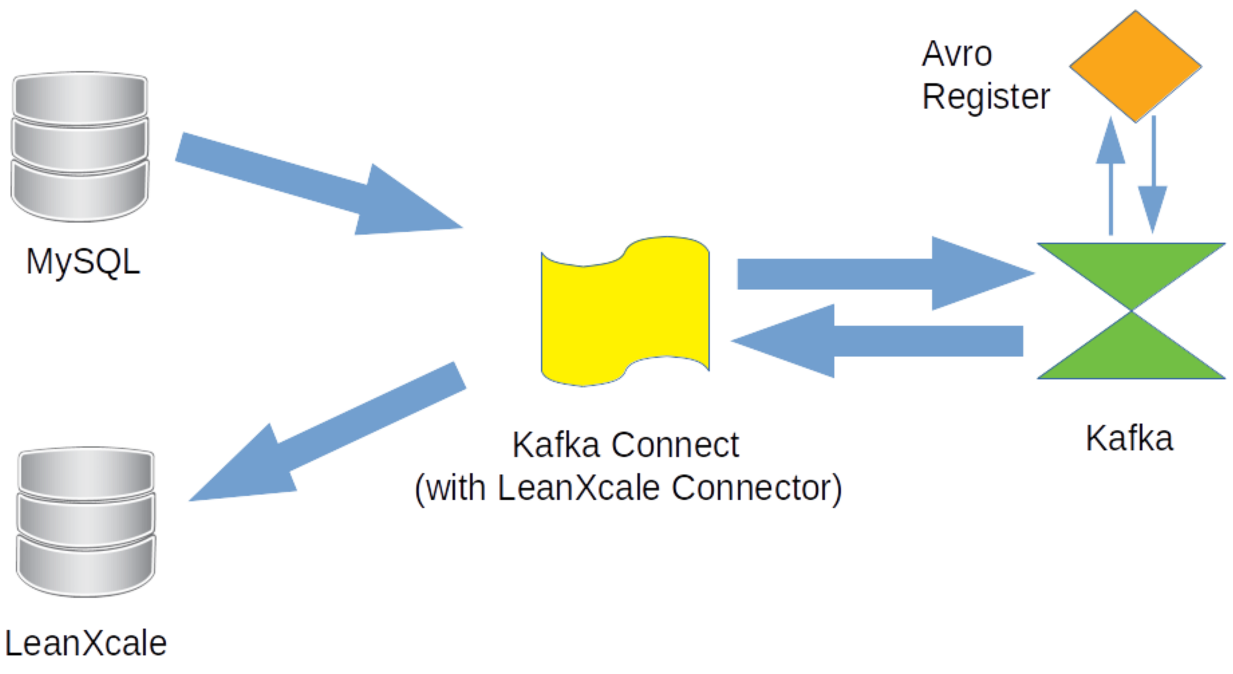

Figure 4: INFINITECH Data Pipeline moving data from MySQL to INFINISTORE using Avro as data serializer

The main difference with the previous architecture is that messages are being serialized using the Avro Schema Registry. The latter allows the messages to contain only the data related information and do not need to replicate schema every time, which is the cases when sending JSON objects without encryption of such tools.

The new configuration for docker-compose includes the new component schema-registry as a docker image:

```
version: '2'
services:
  zookeeper:
    image: debezium/zookeeper:$
    ports:
     - 2181:2181
     - 2888:2888
     - 3888:3888
  kafka:
    image: debezium/kafka:$
    ports:
     - 9092:9092
    links:
     - zookeeper
    environment:
     - ZOOKEEPER_CONNECT=zookeeper:2181
  mysql:
    image: debezium/example-mysql:$
    ports:
     - 3306:3306
    environment:
     - MYSQL_ROOT_PASSWORD=debezium
     - MYSQL_USER=mysqluser
     - MYSQL_PASSWORD=mysqlpw
  schema-registry:
    image: confluentinc/cp-schema-registry
    ports:
     - 8181:8181
     - 8081:8081
    environment:
     - SCHEMA_REGISTRY_KAFKASTORE_CONNECTION_URL=zookeeper:2181
     - SCHEMA_REGISTRY_HOST_NAME=schema-registry
     - SCHEMA_REGISTRY_LISTENERS=http://schema-registry:8081
    links:
     - zookeeper
  connect:
    image: debezium/connect:$
    ports:
     - 8083:8083
    links:
     - kafka
     - mysql
     - schema-registry
    environment:
     - BOOTSTRAP_SERVERS=kafka:9092
     - GROUP_ID=1
     - CONFIG_STORAGE_TOPIC=my_connect_configs
     - OFFSET_STORAGE_TOPIC=my_connect_offsets
     - STATUS_STORAGE_TOPIC=my_connect_statuses
     - KEY_CONVERTER=io.confluent.connect.avro.AvroConverter
     - VALUE_CONVERTER=io.confluent.connect.avro.AvroConverter
     - INTERNAL_KEY_CONVERTER=org.apache.kafka.connect.json.JsonConverter
     - INTERNAL_VALUE_CONVERTER=org.apache.kafka.connect.json.JsonConverter
     - CONNECT_KEY_CONVERTER_SCHEMA_REGISTRY_URL=http://schema-registry:8081
     - CONNECT_VALUE_CONVERTER_SCHEMA_REGISTRY_URL=http://schema-registry:8081
```

For this example, we use a table with different types of fields:

```
CREATE TABLE example (
  id INTEGER NOT NULL PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  description VARCHAR(512),
  weight DOUBLE,
  date1 TIMESTAMP
);
```

Next, the registers are configured. In the source MySQL register, we now include all transformations as well as the Avro converter.

```
{
    "name": "inventory-connector",
    "config": {
        "connector.class": "io.debezium.connector.mysql.MySqlConnector",
        "tasks.max": "1",
        "database.hostname": "mysql",
        "database.port": "3306",
        "database.user": "debezium",
        "database.password": "dbz",
        "database.server.id": "184054",
        "database.server.name": "dbserver1",
        "database.whitelist": "inventory",
        "database.history.kafka.bootstrap.servers": "kafka:9092",
        "database.history.kafka.topic": "schema-changes.inventory",
        "key.converter": "io.confluent.connect.avro.AvroConverter",
    "key.converter.schemas.enable": "false",
        "value.converter": "io.confluent.connect.avro.AvroConverter",
        "key.converter.schema.registry.url": "http://schema-registry:8081",
        "value.converter.schema.registry.url": "http://schema-registry:8081",
        "transforms": "route,unwrap,convert_date1",
        "transforms.route.type": "org.apache.kafka.connect.transforms.RegexRouter",
        "transforms.route.regex": "([^.]+)\\.([^.]+)\\.([^.]+)",
        "transforms.route.replacement": "$3",
        "transforms.unwrap.type": "io.debezium.transforms.UnwrapFromEnvelope",
        "transforms.unwrap.drop.tombstones": "false",
        "transforms.unwrap.delete.handling.mode": "none",
    "transforms.convert_date1.type": "org.apache.kafka.connect.transforms.TimestampConverter$Value",
        "transforms.convert_date1.target.type": "Timestamp",
        "transforms.convert_date1.field": "date1",
        "transforms.convert_date1.format": "yyyy-MM-dd'T'HH:mm:ss'Z'"
    }
}
```

It is important to highlight that in this scenario, we added a new transformation for the Timestamp field in order to send data as a Timestamp type instead of a String format. On the other hand, the LeanXcale sink register remains simple and we only need to add the new fields of the schema and define the Avro Converter:

```
{
    "name": "lx-connector",
    "config": {
        "connector.class": "com.leanxcale.connector.kafka.LXSinkConnector",
        "tasks.max": "1",
    "topics": "example",
        "connection.url": "lx://lx:9876",
        "connection.user": "APP",
        "connection.password": "APP",
        "connection.database": "db",
    "auto.create": "true",
    "delete.enabled": "true",
    "insert.mode": "upsert",
    "batch.size": "500",
    "connection.check.timeout": "20",
    "sink.connection.mode": "kivi",
    "sink.transactional": "false",
    "table.name.format": "$",
    "pk.mode": "record_key",
        "key.converter": "io.confluent.connect.avro.AvroConverter",
    "key.converter.schemas.enable": "false",
        "value.converter": "io.confluent.connect.avro.AvroConverter",
        "key.converter.schema.registry.url": "http://schema-registry:8081",
        "value.converter.schema.registry.url": "http://schema-registry:8081"
    }
}
```

We can now repeat the previous demonstrator, start all components via the *docker-compose* utility, register the two connectors and start ingesting data to MySQL. The Debezium connector will monitor for data modifications, it will send the corresponding transaction logs to Kafka encrypted by Avro and the Kafka connector of INFINISTORE will finally store them.

5.4 Next Steps

After identifying the architecture designs used by modern enterprises in the insurance and finance sector in order to solve common problems and to overcome technological barriers, we defined the notion of the INFINITECH Intelligent Data Pipelines. The benefit of our approach is the use of the *change data capture* paradigm along with the INFINISTORE as the main database management system that provides a variety of innovations that has been described in other deliverables of the project.

Starting the second phase of the project, we continued by implementing our approach and validating its feasibility. As it was mentioned, for our implementation we relied on the Debezium framework that allows monitoring data modification in data sources and propagating these changes using transaction logs via Kafka queues. At the time that this report was written, we have integrated the INFINISTORE with Debezium to be used as the target data source. We have validated the whole implementation using operational datastores as the source, and INFINISTORE as the target. With our approach, we benefit from the innovations and prototypes that have been developed during the first reporting phase and are already delivered and provided by INFINITECH. These are HTAP capabilities of the INFINISTORE, its support for high data ingestion, the Kafka connector and the online aggregates. At this point, we have covered half of what needs to be implemented to completely deliver the INFINITECH Data Pipelines.

In the next period, the focus will be given on implementing pipelines that have the INFINISTORE as the source that needs to send data modifications to other targets. This will be used by scenarios that need to consume data that is firstly stored to the operational datastore of INFINISTORE. Such use cases are the ones that involve the use of the semantic interoperability framework of the project. The latter makes use of an internal triple store and needs to get informed when new data are available. It provides data stream connectors that get data feed for other sources and load the data to the triple store. Based on this, the INFINITECH Data Pipeline can be used and will rely on the technology that is being currently built and delivered under the scope of T5.2 “Incremental and Parallel Data Analytics”. One important outcome of this task is the *incremental scans* that are being currently adopted by the INFINSTORE. This allows the core storage engine of the datastore to propagate data modifications. Having that in place, the next step is to implement the Debezium connector that will monitor the INFINISTORE and send transaction logs to a Kafka queue, using the implementation that will be provided as part of that task. Having that in place, the INFINITECH Data pipelines can also connect to the INFINISTORE with the streaming processing framework of the project, as the latter can also consume data streams coming from a Kafka queue.

Finally, at the time when this report was written, the validation of our approach was based on deployments using the *docker-compose* utility and the testing was performed locally. For the last period of the project, we will provide docker images available through the INFINITECH marketplace and blueprints so that an integrated solution that uses our Intelligent Data Pipelines can be deployed using the INFINITECH way for deployments.

6 Parallized Data Stream Processing using Apache Flink and Kubernetes

In this section we will discuss the second main innovation that will be developed as part of T3.4, namely dynamically scalable stream query processing in distributed query processing environments. More precisely, we first state the motivation and business needs that our work will solve, the problem definition and finally, we give details of the overall design of the system using Apache Flink and Kubernetes. With our design, three key innovations are required to be achieved and we give a more detailed description of the overall system covering all three of them.

6.1 Motivation

In the finance domain, there are a wide range of use-cases that require real-time processing of data streams to add value. For instance, when performing financial trading for currencies or stocks, it is critical to be able to monitor price fluctuations in real-time to identify buy/sell opportunities. Moreover, as the application of alternative information streams such as news articles and social media become more popular, large volumes of specialist compute resources are needed to enable real-time language analytics. However, a key feature of financial data is that the rate at which it arrives at is not constant. Over the course of each day the number of financial trades can fluctuate wildly, and moreover can experience bursts of activity when the market becomes aware of some new information. Indeed, in today's trading environments, the severity of trade burstiness is exacerbated by automatic trading algorithms that use other buy/sell transactions as triggers for their own trades. As a result of these factors, to enable consistent processing of financial data streams with low latencies the underlying infrastructure needs to be elastic to rapid changes in input rate/velocity.

However, elasticity is not a feature currently supported by stream processing platforms such as Apache Flink or Spark. More precisely, such platforms provide what we will refer to as cluster scalability, i.e. compute resources in the form of worker nodes can be dynamically added or removed from the cluster, allowing the total resources available in the cluster to be altered in real-time. On the other hand, the actual stream processing pipeline(s) deployed on the cluster are static, i.e. once configured the resources allocated to them are fixed. Hence, from a practical perspective, these stream processing platforms can't easily tackle data streams with large fluctuations in rate/velocity effectively out-of-the-box.

Instead, what we would want is a platform that provides both cluster scalability and dynamic compute pipelines, where the development of such a platform is one of the aims of T3.4.

6.2 Problem Definition

To formalize the problem being investigated, we are considering stateful multi-operator stream processing applications over bursty data streams, such as currency trading or financial analytics. To clarify the terminology:

- **Stream Processing**: Data points needing processed will arrive over time and need to be processed as quickly as possible as they are used by down-stream components (e.g. user-facing interfaces or machine learned models).
- **Stateful**: One or more operators within the pipeline accumulate state over time that forms a part of the computation performed by those operators. This state needs to be transferred to new instances of that operator during scaling.
- **Multi-Operator**: The pipelines can contain multiple data map, reduce or transformation operations with different performance characteristics.

- **Bursty Data Stream**: The number of data points that arrive on the stream can rapidly vary across different time periods (e.g. by multiple standard deviations).

6.3 System Design

To solve this challenge, the Apache Flink query processing system developed as part of T3.3 and enhanced to enable intelligent data pipelines as described in the previous section, will be further extended to enable dynamic compute pipeline scaling. The core concept for this system is as follows. Given a point in time t, an alert is fired indicating that a pipeline (A) will soon be unable to maintain low processing latencies for a given input stream due to increased load. First, a Flink cluster with more resources will be allocated from the Kubernetes cluster and a new pipeline B will be initalized on it. This pipeline will not receive traffic until pipeline A hits its next checkpoint. When pipeline A receives its next checkpoint the flink source for pipeline A will be disabled, allowing for pipeline A to drain. As the snapshots from each operator are written to the persistent store, pipeline B reads and uses that data to initialize its own operators. Once all operators in pipeline B are initialized then the source for pipeline B is enabled, completing the move between pipelines. At this point pipeline A and its associated resources will be freed on the Kubernetes cluster.

There are three primary innovations required to achieve the above process flow, namely: 1) enabling operator state save/load on demand between replica sets of different sizes; 2) programmatic scaling of Flink clusters on Kubernetes and 3) pipeline configuration transition between clusters. We summarize each on more detail below:

6.3.1 Operator State Saving and Loading

Recent versions of Apache Flink already support checkpointing of operator states via asynchronous barriers on the input stream. Under this model, a central coordinator periodically injects barriers into the data stream, where a barrier represents a point in the stream to effectively take a 'snapshot' of the pipeline. When an operator receives a barrier event, it takes a snapshot of its current state and writes that to a persistent store. If an operator has multiple inputs, it waits/blocks until it receives the barrier from all inputs. This structure assures that the checkpoint meets termination (a complete snapshot will be produced eventually for each input barrier) and feasibility (the snapshot will only include information up-to the barrier) guarantees. This effectively solves the state saving problem, so long as checkpointing is enabled and the snapshots are being written to a secure store then we can recover the state of each operator for different points in the stream.

However, checkpointing within Flink is designed to recover from pipeline failures (e.g. due to a machine failure), not to enable transfer of processing from a low-capacity pipeline to a higher-capacity pipeline. As such, we need to implement a new operator initialization function that enables operators spawned in a new cluster to load state snapshots from equivalent operators in a different existing pipeline. The challenge here is that since different instances of an operator may have distinct state (e.g. because they are processing different subsets of the stream), the load operator needs to understand how partitioning of the stream was performed initially to correctly set state for the new operator instances that use a different partitioning. To explain with reference to Figure 5, if we are transitioning from pipeline A to pipeline B, then for operator 1 the transfer is quite simple, as we simply need to replicate the state from operator 1 in pipeline A to both copies of that operator in pipeline B. However, for operator 2, we move from having two instances (with different states) to three instances, and hence some processing on the instance states needs to be performed to assure that the three new instances in pipeline B have the needed/correct state to function over their new partition of the input stream.

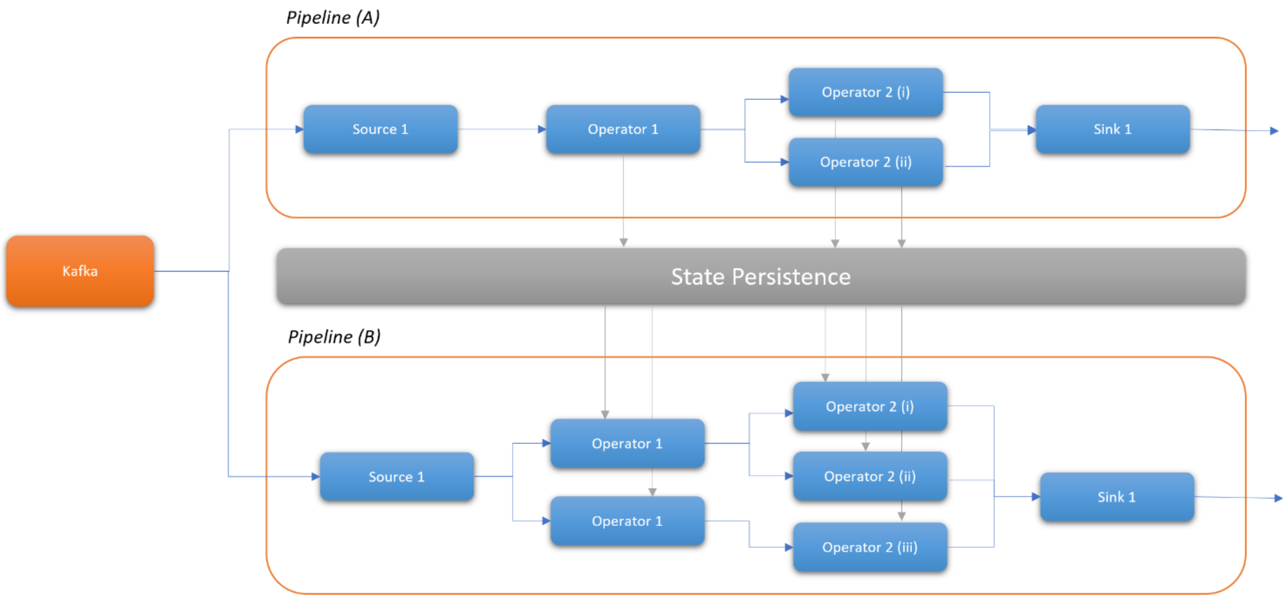

Figure 5: Pipeline Transition Diagram

6.3.2 Programmatic Scaling of Flink Clusters

The second main innovation needed to support scaling of stateful multi-operator Flink applications is the programmatic scaling of the underlying Flink Cluster. Under our core design, we do not scale an existing Flink cluster (although this is possible), but instead allocate a new cluster from first principles with the desired resources, followed by the deletion of the previous cluster when its no longer needed. To achieve this, a separate microservice will be built to facilitate this using the Kubernetes Operator Pattern. Kubernetes Operators are in effect software extensions to Kubernetes to enable automatic management of particular applications and their components. In this case, a Kubernetes operator needs to be developed that is able to:

- Construct docker images encoding a processing pipeline
- Create and configure a new Flink Cluster with a defined total resource allocation
- Delete an existing cluster without loosing the underlying checkpoints of that pipeline

A Kubernetes Operator is itself a separate containerized service with an API that allows functions to be triggered. In this case, one function for each of the three pieces of desired functionality. The operator then communicates with the underlying Kubernetes API to operationalize the changes needed on the physical cluster infrastructure. For example, for our first function, this involves the launching of a Kubernetes Pod that executes the Docker build process to construct and then upload a container image containing our Flink compute pipeline to a docker image repository (which can be later used to produce a new Flink cluster pre-loaded with the desired compute pipeline).

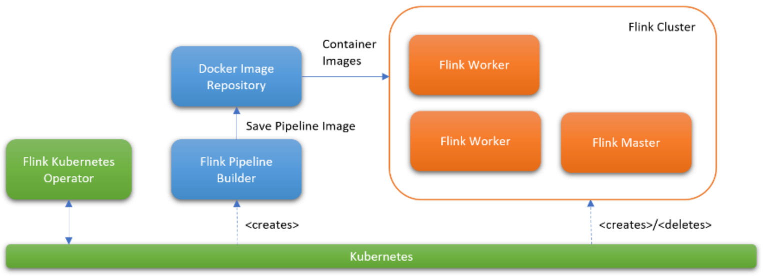

Figure 6: Flink Operator and Flink Cluster Creation

6.3.3 Pipeline Configuration Transition between Flink Clusters

The final innovation that is needed to enable scaling of stateful multi-operator Flink applications is a controller service to both trigger and then manage the overall transfer between Flink pipelines. In effect, this control service needs to provide the following functionality:

- Rule-based Predictive Pipeline Failure Identification: The ability to define a set of rules that take as input a recent set of time-series metrics exported by a compute pipeline and identify whether that pipeline needs to scale up or down. Internally, the service will regularly evaluate the different user-defined rules to see if any are violated. If so, a pipeline transition will be started.
- Pipeline Creation: If a pipeline transition is triggered, then the first action the controller service needs to perform is the creation of a new Flink cluster with appropriate resources. This is achieved through communication with the Flink Operator service defined in Section 6.3.2. Note that this operation may take some time, and so this process needs to block until the operator reports the pipeline is in a running state.
- Data Stream Redirection: Once the new pipeline is operational, the data stream needs to be re-directed to the new processing pipeline.
- Pipeline Deletion: The final step in performing a transition between pipelines is removing the previous pipeline, via communication with the Flink Operator service defined in Section 6.3.2.

7 Conclusions and next steps

This document reported the work that has been done in the scope of task T3.4 “Automated Parallelization of Data Streams and Intelligent Data Pipelining”, whose objective is twofold. First, to provide the enablers for deploying intelligent data pipelining, thus having what we call the INFINITECH approach for intelligent data pipelines. This will provide a holistic solution that addresses all problems that currently appear in different architectural designs used in the modern landscape. The base will be the innovations brought by the data management layer of INFINITECH, which solves the problem of data ingestion in very high rates, removing the need for database offloading, along with the *online aggregates* of the declarative real-time analytical framework of INFINITECH. This removes all issues having to pre-calculate the results of complex analytical queries, which leads to inconsistent and obsolete results. The integration of INFINISTORE with Apache Flink, as part of the work being currently done under the scope of T3.3 “Integrated Querying of Streaming Data and Data at Rest” and the integration with tools for Change Data Capture (CDC) that will done under the scope of this task, will enable the deployment of such intelligent data pipelines.

The second objective of task T3.4 is to provide the means for enabling automated parallelization of data streams, allowing to dynamically scale out individual operators that formulate a data stream in order to cope with diverse incoming workloads. Current solutions allow static deployments of stateful multiple operators, but once deployed, they cannot be scaled out. Our design allows operators to save and load their state, which allows to shutdown existing deployments and redeploy them increasing their instances, while transmitting the state of the formers to the new ones. The use of Kubernetes as the underlying container-orchestration system for automated deployments allows to programmatically scale the Apache Flink clusters and our novel operators allow to the configuration of the transmission of the state across those clusters.

At the first phase of the project, the main focus was given in implementing and delivering the baseline technologies that will create the innovation and break through the current barriers of modern organizations that require real-time processing and analytics over multiple data sources (either static *at-rest* or streaming and *in-flight*). That is the HTAP provision, which enables analytical query processing over live operational data without the need to move snapshots of a dataset to a data warehouse, the capability of the data management layer to allows for high rate data ingestion via its dual interface, its polyglot extensions that allows query processing over federated datastores and finally, the *online aggregates* using declarative scripting language, ensuring data consistency at the same time in terms of database transactions. As all these technologies have been incorporated into the INFINISTORE, the integration of the latter with Apache Flink as the baseline technology for the INFINITECH streaming processing framework was the second necessity. The automation of its deployment using container-orchestration frameworks now allows the automated parallelization of the data streams, which is the second target objective of this task. The delivery of those fundamental pillars was the primary focus during the first phase of the project, with the first prototypes being now already available.

Moreover, during this first phase of the project, an intensive analysis of the state-of-the-art of streaming processing frameworks took place as well, allowing us to identify the current architectural designs of the modern landscape, along with their inherit barriers. After conducting this analysis, we defined the vision of the outcomes of this task, which gave valuable input to the rest of the tasks related with the data management activities of INFINITECH and more precisely, T3.1, T3.2, T3.3 and T5.3. As these tasks have been progressed and the first prototypes have been already delivered, we are now in a position to start the implementation of our holistic solution in what we call the INFINITECH approach for intelligent data pipelines. Additionally, a thorough analysis on how the innovations developed so far will solve all aforementioned barriers of current architectural decisions have been provided. Moreover, the initial design on how to allow the automated parallelization of data streams has been included in this report. Our design allows to dynamically scale out individual operators of the data stream, transferring the current state of the deployed Flink cluster to the new ones, thus, removing the barrier of having to rely on static deployments that cannot cope with diverse workloads in real time.

Having the basic pillars and design in place, during the second phase of the project task T4.3, we provided a first implementation of our INFINITECH Data Pipelines. We validated our approach by using the *change data capture* paradigm to transparently move data from external datastores that can be considered as *sources* to the INFINISTORE. We utilized a commonly used commercial operational datastore as the *source* and we setup and deployed the intelligent data pipeline so that data ingested in the operational data store can be stored and retrieved from the INFINISTORE. That way, we allow the data analysts and application developers to benefit from exploiting the innovations developed within INFINITECH and integrated into its data management system, in order to avoid the need for hybrid and complex architectures. As we saw from our analysis, to maintain such architectures that involve numerous and heterogeneous database systems can be a hard task while on the same time, all different approaches come with their inherit drawbacks and technological obstacles. In the final phase of the project, we will validate a data pipeline whose *source* will be the INFINISTORE itself and the target other data management systems. This is planned to be exploited by the integration of the Semantic Interoperability Engine of INFINITECH with the INFINISTORE, as explained in section 5. Finally, during the last phase, more details along with additional demonstrators will be given regarding the automation of parallelized data streams and how our implementation can scale out the streaming nodes in real-time by storing and replicating their internal state.