


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



## D3.8 – Data Streaming and Data at Rest Queries Integration – III

<b>Revision Number</b>	3.0
<b>Task Reference</b>	T3.3
<b>Lead Beneficiary</b>	LXS
<b>Responsible</b>	Ricardo Jiménez-Peris
<b>Partners</b>	LXS, GLA, UBI, UPRC
<b>Deliverable Type</b>	Report (R)
<b>Dissemination Level</b>	Public (PU)
<b>Due Date</b>	2021-12-31
<b>Delivered Date</b>	2022-03-11
<b>Internal Reviewers</b>	ASSEN, FTS
<b>Quality Assurance</b>	INNOV
<b>Acceptance</b>	WP Leader Accepted and Coordinator Accepted
<b>EC Project Officer</b>	Beatrice Plazzotta
<b>Programme</b>	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 <sup>1</sup>	Author(s) <sup>2</sup>
LXS	Lead Beneficiary	Ricardo Jiménez-Peris
LXS	Contributor	Boyan Kolev, Pavlos Kranas, Alejandro Ramiro, Spencer Pablos, Javier Pereira
GLA	Contributor	Richard McCreadie, Craig Macdonald, Iadh Ounis
UBI	Contributor	Konstantinos Perakis, Dimitris Miltiadou Stamatis Pitsios
UPRC	Contributor	Christos Doulkeridis, Ioannis Kranas
ASSEN	Reviewer	Ilesh Dattani
FTS	Reviewer	Juergen Neises
INNOV	Quality Assurance	John Soldatos

## Revision History

Version	Date	Partner(s)	Description
0.1	2022-03-03	LXS	ToC Version
0.2	2022-03-04	All	Input in section 6 and 4
0.3	2022-03-04	LXS	Update the intro and conclusions
1.0	2022-03-04	LXS	Finalize the document for internal review
1.1	2022-03-07	ASSEN	Internal review
1.2	2022-03-07	FTS	Internal review
2.0	2022-03-07	LXS	Submission for internal QA
3.0	2022-03-09	INNOV, LXS	Final submission after internal QA

<sup>1</sup> Lead Beneficiary, Contributor, Internal Reviewer, Quality Assurance

<sup>2</sup> Can be left void

## Executive Summary

The goal of task T3.3 “Integrated Querying of Streaming Data and Data at Rest” of the INFINITECH project is to implement a data framework that can provide a unified manner for accessing data that can be considered both *streaming* and *data-at-rest* at the same time, while being able to correlate data coming from those different types of data sources. This data framework aims to overcome the existing obstacles that are observed in current solutions. Even if currently available solutions state that they enable the provision of real-time business intelligence (BI), they often provide something *near* real-time due to the inherent limitations of the tools they rely on. The important challenge that the INFINITECH unified query framework aims to solve is to provide actual real-time BI that is crucial in a variety of use cases that the INFINITECH platform supports, such as real-time risk assessment, transaction fraud detection, money laundry, etc.

The INFINITECH unified query processing framework relies on the Apache Flink, one of the popular streaming processing tools, extending it with SQL operators that enables the correlation of streaming data with data *at-rest*, removing the barriers for real-time processing. This correlation is achieved by reading data from the platform’s data management layer and performing cost-demanding analytical operations in a cost-effective manner that can be used as a streaming operator, or by allowing the data ingestion of streaming data to the persistent storage, modifying its content while at the same time, ensuring transactional semantics. Towards this direction, this task has exploited the outcomes of other tasks related with the data management layer of INFINITECH, among those the ultra-scalable transactional management, its incremental analytics and the Hybrid Transactional and Analytical Processing (HTAP) provision, the declarative online data aggregations, and potentially the polyglot extensions of the platform. The outcomes of those tasks constitutes the basic pillars have been utilized by the operators implemented in the scope of this task, which can now allow the unified query processing framework to provide real-time BI.

This deliverable describes the steps required for the INFINITECH unified query processing framework design and implementation. At the first phase of the project, an initial analysis of the state-of-the-art in the field of data streaming processing had been conducted in order to decide which of the proposed solutions would be better suited to be used as the core of the framework. Based on this decision, an initial design of the operators that now extends the proposed data streaming processing was made, which drove the actual implementation during the second phase of the project. This was necessary as those operators rely on the outcome provided by other technical tasks of WP3 and WP5, leading the implementation to be initiated at the second phase of the project (M12-M27). In the second version, the design of the integrated solution of pilot#2 took place in order to benefit from the outcomes of the work that was being currently carried out under the scope of the task T3.3. In this last version of this deliverable we document the final implementation of the operators provided by the INFINISTORE Flink Connector that allows the combination of static and streaming data, taking advantage of the outcomes of the technical tasks of WP3 and WP5, along with the experimentation with the aforementioned pilot.

# Table of Contents

1	Introduction.....	7
1.1.	Objective of the Deliverable.....	9
1.2.	Insights from other Tasks and Deliverables.....	9
1.3.	Differences with the previous version of this report .....	10
1.4.	Structure.....	10
2	Relation with INFINITECH use case scenarios .....	11
2.1	Problem Dimensions.....	11
2.2	The Case of Real-time Risk Assessment in Investment Banking .....	12
3	State-of-the-Art Analysis on Data Streaming Technologies and Complex Event Processing.....	14
3.1	Data Streaming Technologies and their Generations .....	14
3.1.1	Apache Storm .....	15
3.1.2	Apache Spark.....	15
3.1.3	Apache Flink.....	15
3.1.4	Amazon Kinesis.....	16
3.1.5	Apache Samza.....	16
3.1.6	IBM InfoSphere Streams.....	16
3.1.7	Brief Comparison of Analysed Data Streaming Tools.....	17
4	INFINITECH Enablers for SQL Operators over Streaming Data.....	18
4.1	Hybrid Transactional and Analytical Processing.....	19
4.2	Online Aggregations .....	19
4.3	Polyglot Capabilities .....	20
4.4	Incremental Analytics .....	20
5	INFINITECH streaming engine overview and design of operators .....	22
5.1	Basic Concepts.....	22
5.2	Stream Correlation with Data At-Rest.....	24
5.3	INFINITECH Operators for the Streaming Engine .....	25
6	The INFINISTORE Flink Connector .....	30
6.1	The INFINISTORE Flink Connector: The core .....	30
6.2	The INFINISTORE Flink Connector: The sink implementation .....	32
6.3	The INFINISTORE Flink Connector: The source implementation.....	33
6.4	Implementation of Flink source abilities .....	36
6.5	Demonstration of INFINISTORE Flink Connector.....	38
7	Combined Data Streaming and Data at Rest Illustration .....	45
8	Conclusions.....	47
9	References .....	50

## List of Figures

Figure 1: Example of the VaR calculation process for FX data streams .....	13
Figure 2: Flink Connector – The Core Factory .....	31
Figure 3: Flink Connector – The Sink implementation .....	32
Figure 4: Flink Connector – The Source implementation .....	34
Figure 5: Flink Connector – The Source implementation: KiviSourceFunction .....	35
Figure 6: Flink Connector – The Source implementation: abilities .....	36
Figure 7: Flink Connector – The Source abilities: SupportsFilterPushDown .....	38
Figure 8: Naive VaR Flink Topology .....	45
Figure 9: VaR Calculation Topology using the Infinitech Streaming Engine .....	46

## Abbreviations/Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
BI	Business Intelligence
CEP	Complex Event Processing
CQL	Continuous Query Language
CPU	Central Processing Unit
DoA	Description of Action
DL	Deal Learning
ETL	Extract, Transform, Load
HTAP	Hybrid Transactional and Analytical Processing
I/O	Input/Output
IoT	Internet of Things
JDBC	Java DataBase Connection
ML	Machine Learning
NoSQL	No/Not only SQL
RDD	Resilient Distributed Dataset
SDG	Stateful DataFlow Graph
SQL	Structured Query Language
VaR	Value at Risk
WP	Work Package

# 1 Introduction

Finance and insurance institutions utilize static data that are persistently stored in a database management system, often called as *data-at-rest*, in order to extract information via analytical tools and AI algorithms that rely on historical data. Therefore, their analysis is executed as a batch process, once the tool or algorithm is being invoked, relying on the data that are persistently stored in the data store at that exact point in time, which does not always reflect the situation at the same point in time. In addition, as explained in D3.1, organizations that utilize Big Data tend to use Extract, Transform, Load (ETLs) periodically in order to move data from their operational data stores to a data warehouse, where they perform their analytics. As a result, the latter makes use of a snapshot of the dataset that was taken at the specific point in time when the ETL process moves the data to the warehouse. This can pose an issue in cases where an enterprise needs to be aware of potential risks or opportunities in order to adapt and exploit them at the time when they happen. In the finance and insurance sectors there are many cases where the time window to perform an action is narrow and performing analysis on yesterday's data can hinder effective courses of action. Such examples in the finance sector include risk assessment analysis, where a financial organization might need to provide detailed risk information regarding the management of an asset in real-time, otherwise an investment opportunity could be lost. Another example can be noticed in fraud detection mechanisms, where the identification of a fraud transaction must be done exactly the moment when the transaction takes place, since analyzing the historical transactions of the previous day could prove ineffective. Moreover, in the scope of the insurance sector, taking IoT sensor data coming from devices, either from vehicles or from people's smart phones could prove crucial to occur in real time since utilizing historical data could result in losing the opportunity to extract vital information at the time that the data are produced. Those scenarios are observed often in financial institutions and the insurance sector and pose typical challenges to many of the organizations of these sectors. They are also listed as typical user requirements from the pilots of the INFINITECH project, as they have been addressed in the corresponding deliverables of T2.1.

Due to the need of real-time data analytics, streaming processing systems have been widely used during recent years. The emergence of IoT, where data are being continuously produced by various sources (either a hardware sensor that is physically installed or data generated after an online transaction) has led to organizations having different types of streams being accessed by their systems. In order to utilize this new types of data, various data streaming infrastructures have been developed that allow application developers and data analysts to perform some query processing on top of the stream. In contrast with traditional database management systems where data are persistently stored and considered *at-rest*, where queries are submitted dynamically and produce results in a request-response manner, the nature of the streaming processing is different. Queries are statically submitted and make use of dynamic data (coming from the stream) often called data *in-flight*, and thus, they are considered continuous. As queries are not dynamic, there is no request-response type of interaction, rather than once a continuous query has been submitted, it continuously generates results. Queries might be stateless where no previous information might be needed. Examples can be found in scenarios where a financial organization needs to check if the amount of an online transaction exceeds a specific threshold. In case it does, this event might trigger additional actions from the organization to examine the transaction details and potential fraud activities. Typically, those queries only require comparison of the current data coming from the stream with a static value. However, as data is being processed in real-time, it allows the financial institution to react instantly, without having to perform this type of analysis on obsolete data coming from a snapshot taken in the past. Additionally, continuous queries might be stateful and require some timestamp information that has been collected from data being passed through the stream channel previously. Usually, a *time window* is being maintained that allows for aggregated operations to take place. An example will be to produce an alert if the value of a data element coming through the stream is bigger than the average value of all data elements that have been passed during the last minute, hour, etc. This reveals potential current trends and might be useful in scenarios, for example, where a lot of investors choose to buy a specific stock or other investment product, or if clients decide to massively withdraw money from their accounts, or move their

money to other products, which might be the case of a *bank run* due to a potential currency devaluation as the global economic crisis that started in 2008 showed. The streaming processing framework calculates the aggregated values of money transfer in the last minute or hour and might generate an alert to the financial institution in case massive money transfer occurs. It is obvious that if the financial institution had to rely on a periodic batch processes using other types of analytical tools on an obsolete dataset, the results might be catastrophic for the institution which could face hazardous liquidity issues.

Those two different types of processing, data at-rest and streaming data, can solve different types of problems addressed by the finance and insurance sector. Dynamically submitted queries at data *at-rest* can feed machine learning (ML)/Deep learning (DL) algorithms taking into account all historical data stored in a persistent medium like a data warehouse, but cannot record changes or trends happening in real-time. On the other hand, static continuous queries can generate events to which an organization can respond immediately. However, they can only rely on a narrow time window and cannot take into account the existing historical data. One of the current challenges arising during recent years is the ability for query processing that involves both worlds: data at-rest and streaming data. This can enable real-time business intelligence (BI) where:

1. streaming processing can be combined with the results of an analytical processing or
2. streaming data can be directly ingested in a data warehouse, and the AI algorithms can rely on fresh data.

However, both approaches come with their limitations and can only provide *near* real-time BI due to various inherent obstacles. In order to collate streaming processing with aggregate/analytical queries targeting data stored in a database, it requires the latter to be executed first, get the result, and compare the result with the streaming data element on the fly. However, aggregate and analytical queries on a dataset need a *scan* operation, meaning that the majority of a dataset must be accessed first. Typically, these types of queries are costly, and therefore, cannot be used in streaming processing, where the latency must be very low. To overcome this inherent obstacle, traditional approaches often *cache* those results in memory and periodically update their values. This breaks however the data consistency, which is of major importance in the financial sector, as data is outdated. Ingesting data from a stream to a persistent storage and performing analytical queries in the datastore itself, comes with other obstacles. Traditional database systems cannot handle such an increased operational workload coming from a data stream, as they cannot scale out effectively. Due to this, system architectures either rely on NoSQL database systems, losing however transactional semantics and data consistency, or tend to add data coming from a stream to a data queue, and then periodically perform batch ingestion on the database. The latter approach leads to the use of *near* real-time BI, while the pilot use cases of INFINITECH aim to go a step beyond and do analytics on data, as they arrive.

The task T3.3 “Integrated Querying of Streaming Data and Data at Rest” aims to solve the aforementioned problems: providing a unified framework that allows application developers and data analysts to perform analytics considering elements coming from both worlds. This means correlating streaming tuples with data at-rest in both ways: reading from a persistent storage and correlating the results with data coming from a streaming channel and using data streams to update and modify the contents of a persistent datastore. It will use the current advancements in the technology provided by the INFINITECH project, and most precisely the ability of its datastore, the INFINISTORE, to support massive data ingestion, its polyglot capabilities, and its enablers for advanced analytics: the *online aggregates* and the *incremental analytics*.



## 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 T3.3 at the end of its duration (M27). This report consist of three versions, each one of those extends and modifies when necessary the content of this document, following the agile approach for system development and aiming to update the solution and implementation with the current trends of the environment as the project is progressing. The work that has been done during the first two phases (M03-M20) was mainly focused on the experimentation of various streaming processing frameworks that are currently being used in the industry, in order to decide which of those engines the INFINITECH unified query processing framework will rely on. Based on this preliminary work that was essential for this task, the initial design of the data operators that will allow the correlation of data elements from both worlds took place. The correlations themselves are relying on the outcomes of the various tasks that are related with the data management layer, as their implementations provide the basis for the implementation of the task T3.3 to happen.

In the previous second version of this document, T3.1 (“Framework for Seamless Data Management and HTAP”) and T5.3 (“Declarative Real-Time Data Analytics”) had already produced their first prototypes and as a result, the implementation of the operators designed in the first version of the this report has already started. Therefore, an initial work had been conducted in order to allow the experimentation of the outcomes of this task with an actual use case coming from pilot#2 of the INFINITECH project. In this last version of this document, the outcomes of T5.2 (“Parallel and Incremental Analytics”) have been also delivered, which allowed us to provide the final delivery of the implementation that took place this last period and is reported in the last version of this deliverable. It concerns the data operators provided by the INFINSTORE to the streaming processing framework that allows the latter to take the full advantage of the newly adapted technologies.

## 1.2. Insights from other Tasks and Deliverables

As the majority of the deliverables of WP3, the work that is reported in this document is based on the overview description of the corresponding task T3.3, which has been further specified in more detail at WP2, which is the fundamental work package that defines the overall requirements of the whole platform. T2.1 defines the user stories of the pilots that drive the necessity of this task, while T2.3 defines the specification of the overall technologies that INFINITECH provides and need to interact with the unified query processing framework. T2.5 describes the available datasets that need to be tackled by this component while T2.7 puts the component under the general context of the INFINITECH Reference Architecture. Regarding the technical tasks of the project, T3.3 is relying on scalable transactional processing of the INFINITECH data management layer, as described in the corresponding deliverables of T3.1, along with the Hybrid Transactional and Analytical Processing (HTAP) capabilities that this task provides. Moreover, as explained in the corresponding deliverables of T3.2, the polyglot processing is an extension of the data management layer, and therefore, this task can exploit its outcomes in order to correlate streaming data with data stored in external data sources, in the case of scenarios where data cannot move inside the deployed INFINITECH sandbox. In addition, T3.3 will also exploit the outcomes of T5.3 and its declarative live aggregation mechanisms that will allow the execution of cost-demanding aggregation operations with  $O(1)$  complexity, and hence, making it possible to correlate streaming data with this type of information. Moreover, with the latest advancements of T5.2, this task will exploit the provision of incremental analytics currently adapted by the INFINSTORE to allow the streaming processing framework to submit un-bounded queries that can return results in a new data stream that can be further processed by the streaming framework. Finally, T3.3 provides valuable input to T3.4, whose scope is to provide automated parallelization of data streams that will rely on the operators implemented in this task.

## 1.3. Differences with the previous version of this report

The main difference of this last version of the deliverable with the previous one is the addition of the *Incremental Analytics* subsection of chapter 4, where we document an additional INFINITECH enabler or the integrated query processing. This allows for the submission of un-bounded query statements to the INFINISTORE, whose result can be now retrieved over a data stream. Moreover, in this last version we include chapter 6 that provides all the design and implementation details of the proposed INFINISTORE Flink connector that has been developed during this last phase of the task. An additional subsection demonstrates its use via the documentation of various test cases that have been created to validate its functionality. Finally, we have extended the section that states the *conclusions* to also include how the outcomes of this task have accomplished its objectives the relevant KPIs of the INFINITECH project, as defined in the DoA.

## 1.4. Structure

This document is structured as follows: Section 1 introduces the document and section 2 provides concrete examples on how the outcomes of T3.3 can be utilized by the pilots of the INFINITECH project. Then section 3 provides the state-of-the-art analysis of existing solutions for complex event processing and data streaming frameworks, highlighting the existing barriers of those solutions to provide real time business intelligence. Section 4 analyses how the technological achievements of INFINITECH can be used as enablers to overcome those barriers. This section has been extended in this last version of this document to include the newly delivered advancements of the incremental analytics that are now used by the integrated streaming processing framework. Based on the output reported in previous sections 3 and 4, section 5 describes the design of the SQL operators that will be implemented in order to extend the streaming processing framework to correlate streaming data with data at-rest.

An additional section 6 is now included in this last version of the report, documenting the implementation of the project's streaming connector that includes all operators that had been previously designed in the previous phases of the project. It describes its core, its *sink* and *source* part, how it allows the streaming processing framework to push operations down to the storage engine, along with a demonstrator of its use.

Section 7 now provides an illustration of how the outcomes of the task T3.3 would add value with respect to a specific INFINITECH Pilot, pilot#2, and how the latter can benefit from technology and innovation that is being created from the work that is being currently carried out under this task.

Finally, section 8 concludes the document and addresses next steps.

## 2 Relation with INFINITECH use case scenarios

In the fast-moving financial domain, it is critical to maintain up-to-date analytics over financial markets. Such analytics are used by a wide range of both human and AI traders continuously throughout each day. However, in practice, these analytics are not as simple as tracking a stock or share price. Instead, more complex metrics are needed that compare real-time market changes with long-term historical trends, whilst also incorporating the current position/exposure of the individual trader. For example, for the purposes of Pilot#2 (“Personalized Retail and Investment Banking Services”) a common metric used by financial traders around the world is Value at Risk, which measures the potential risk of the trader’s current investments by analysing the variance of the associated assets over different time horizons (that can involve years’ worth of data points).

Metrics like Value at Risk raise a number of computational challenges, as they require both real time market data streams (*data-in-flight*) to remain current, but also need large quantities of historical data (*data-at-rest*) to provide meaning in context. Decades of research into stable database solutions have produced a range of good quality products to manage data-at-rest, including MongoDB, MySQL, PostgreSQL, LeanXcale, among others. Meanwhile, although less mature, a number of streaming platforms for processing data-in-flight have been under development over the last 10 years, such as Apache Storm or Apache Flink. However, the architectures of such databases and streaming platforms are very different and are not designed to be compatible. Hence, developing applications that require seamless integration of both databases and streaming platforms is very difficult and requires significant specialized expertise.

Task 3.3 in general aims to make such integration easier for the applications within the financial domain, by producing a framework for orchestrating the aggregation of data-in-flight (streaming data) and data-at-rest (i.e. historical data within an SQL database).

### 2.1 Problem Dimensions

It is first worth noting that there is a large space of possible ways that an application developer might want to aggregate data-in-flight with data-at-rest. For example, a streaming select involves taking each item that arrives on an input stream and performing a SQL SELECT operation for that item, before sending both the item and the query return on an output stream. Meanwhile, a windowed timeseries function involves periodically performing a local analytics function on a small streaming data window, writing the result to a database, and then retrieving the window timeseries for a longer time period to calculate an aggregate measure. Furthermore, the appropriate solution will also be based on a range of environmental factors, such as whether it is possible/desirable to continuously store the incoming data streams, the available compute capacity that can be allocated to each individual stream, along with expected storage and network latencies. Hence, to better organize T3.3, we structure the problem along the following dimensions:

Table 1: Problem Dimensions

Dimension	Value	Description
Processing Type	Per-Item	The user is looking to augment an item that has arrived on a stream with associated data in a database
	Windowed	The user is looking to perform calculations over a series of time windows, where a subset of those windows is stored in a database
Outcome Writing	True	Once the calculation is finished, the outcome

		needs to be stored in a database
	False	The calculation is read-only on the database
Stream Writing	True	The raw contents of the data stream will be stored in a database
	False	The stream contents are not stored
Computation Locality	Streaming Platform	All significant computation is performed within the streaming platform (the database is used only for basic data lookup)
	Streaming Platform & Database	Computation is shared between the streaming platform and database

Importantly, developing a technology that is able to solve all dimension combinations is out of the scope of T3.3. Instead, we focus on a sub-set of dimension combinations that align with the INFINITECH pilots that require such a technology, which we discuss in the next section.

## 2.2 The Case of Real-time Risk Assessment in Investment Banking

The high-level aim of this case, which is further analysed in Pilot#2, is to provide bank traders real-time information about financial assets they may wish to trade, ultimately enabling improved decision making and hence profit margins for their customers. Currently, trading information and future predictions are updated infrequently (once a day), meaning that traders are unable to exploit rapidly changing market conditions. This case should solve this issue by providing a solution that can aggregate market data, trends and provide predicted risk/yield estimates that update in real-time.

Within the wider trading platform that this case supports, one component that requires data-in-flight and data-at-rest to function is asset risk estimation. The goal of this component is to monitor the stream of financial asset costs and the current exposure of the trader to those assets (i.e. how much the trader has invested), and then calculates a range of risk metrics. This is used to help traders track the short and long-term risks of particular investments or their broader portfolio.

To illustrate, we will use the example of one very common metric, Value at Risk (VaR). The aim of VaR is to determine the potential loss for an asset and the probability that the loss will occur. The primary input to VaR is the return on an asset (how profitable it is) over time. This is expressed by a very large numerical timeseries spanning millions of data points per year for each asset. This is combined with various parameters, such as the target time period to calculate VaR over, as well as the current exposure of the trader. The calculation of VaR (and similar metrics) is costly, particularly when calculating for long time periods with high datapoint counts, or if performing a significant forward projection. Moreover, if the underlying return for an asset changes rapidly, then the trader will want to be notified of the increased VaR (i.e. estimated risk) with very little latency, such that they can take remedial steps. However, constantly re-calculating VaR from first principles for potentially hundreds of thousands of assets is not feasible.

On the other hand, it is possible to substantially reduce the cost of the VaR calculation through incremental calculation of its constituent components (the mean and standard deviation of the asset return timeseries) across smaller time windows. This can be achieved using a streaming platform that buffers datapoints into fixed time windows and triggers processing once each window is full. However, a streaming platform itself cannot safely store the resultant intermediate outcomes as they are designed to be stateless. Hence, the intermediate outcome from each window needs to be stored into a database and made accessible such that VaR can be rapidly calculated for a target time period. The challenge then from the database side is to

provide sufficiently fast writes for the new windows as they are created, while also enabling very low-latency querying of the stored window data for each asset such that VaR can be re-calculated for tens of thousands of assets minute-by-minute.

Figure 1 illustrates the structure of this process for FX assets (currency trading) when combining data-in-flight and data-at-rest. As we can see, a high-volume data stream of currency prices continually arrives at the left hand side, comprised of <asset, timestamp, value> tuples. This stream is then sub-divided into one stream per-asset. The streaming platform will then buffer the updates for each currency price into fixed time windows (in this example a 5 minute window length). Once the buffer period has elapsed, a trigger starts the calculation of the intermediate components needed for VaR, i.e. the mean and standard deviation of the updates within the window, that are then stored within a database. Depending on the desired variants of VaR the user wants, the required window data is loaded from the database and those variants of VaR are calculated and then emitted for downstream consumption by the user.

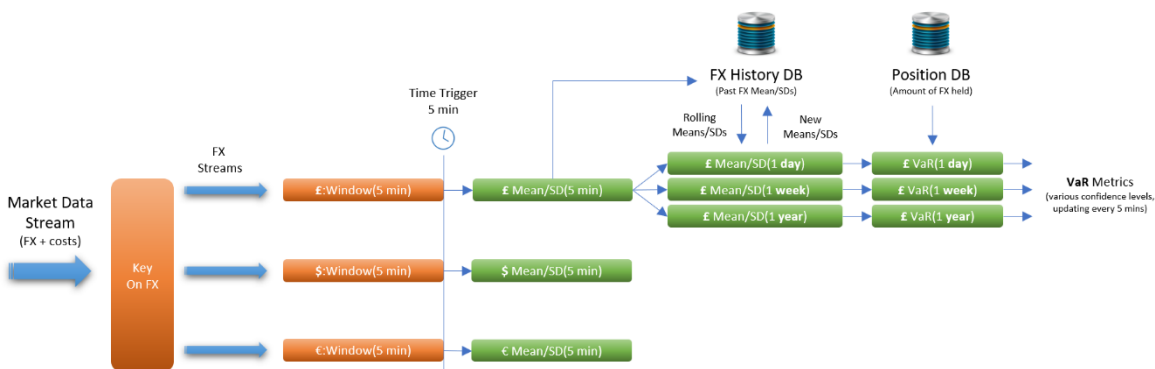


Figure 1: Example of the VaR calculation process for FX data streams

Considering the dimensions discussed earlier, this is a windowed process, i.e. it relies on intermediate calculation over time windows, and the computation is primarily performed within the Streaming Platform. Meanwhile, calculation outcomes are being written (means and standard deviations), but the raw stream is not.

This is a proposed solution for a general case application coming from the insurance sector targeting risk assessment analysis in real-time. However, we designed it in a more generic manner in order to apply the same patterns to other scenarios coming from both the insurance and finance sector. INFINITECH has 15 pilot cases, where many of them require streaming processing technologies, as identified in the user stories provided by T2.1 Therefore, we will take advantage of those pilots during the evaluation phase to verify if the proposed framework that we present in this deliverable can be beneficial to them.

## 3 State-of-the-Art Analysis on Data Streaming Technologies and Complex Event Processing

Big data analytics is a key area for businesses and the public sector alike, enabling the analysis of huge amounts of data to draw business insights and discover new, innovative ideas, technologies and solutions. By utilizing big data analytics and artificial intelligence, businesses and organizations can support their BI, adopting new data driven decision-making tools and shifting strategic planning processes.

The data collected by the various systems day by day are rapidly increasing and that makes it difficult to store them in known relational and non-relational business' databases but also, to apply data mining tools and techniques directly on big data streams. Thus, nowadays it seems that streaming data processing, the technology that started to develop more than 20 years ago, has a greater value than ever. In 1992, the Tapestry system has introduced the notion of streaming queries, and by then, various technologies of streaming processing have been developed per generation of streaming systems.

### 3.1 Data Streaming Technologies and their Generations

Starting with the first generation, the applications that have been developed in the early 00s used centralized stream processing engines such as Stream [1], Aurora [2] and TelegraphCQ [3]. These engines provided window-based query operators that execute continuous queries over relational data streams. While these engines supported principled relational query models, e.g. as proposed through the Continuous Query Language (CQL) [4], they lacked support for parallel data processing, making them inapplicable in Big Data scenarios.

With the increase of stream rates and query complexity, a second generation of stream processing engines became distributed in order to harness the processing power of a cluster of stream processors. Systems such as Borealis [5], Gigascope [6], and InfoSphere Streams [7] permit inter-operator parallelism for continuous queries, that is, one query can be executed on multiple machines. Such systems exploit task-parallelism, i.e. they execute different operators on different machines and allow the execution of many different continuous queries in parallel. InfoSphere Streams supports intra-query parallelism through a fine-grained subscription model, which specifies stream connections, but management is manual.

As a result, the third generation of stream processing engines focus on intra-query parallelism, parallelizing the execution of individual query operations. StreamCloud [8], Apache S4 [9] and Storm [10] express queries as directed acyclic graphs with parallel operators interconnected by data streams. StreamCloud parallelizes stateful queries at runtime also providing intra-operator parallelism. It uses a query compiler to synthesize high-level queries into a graph of relational algebra operators. StreamCloud also provides elasticity. It uses hash-based parallelization, which is geared towards the semantics of joins and aggregates. S4 schedules parallel instances of operators but does not manage their parallelism. Storm allows users to specify a parallelization level and supports stream partitioning based on key intervals, but it cannot scale out the computation at runtime. This makes it hard to support unknown Big Data analytics tasks when the computational expensive of operators is not known beforehand. Schneider et al. [11] adds elastic operators to the SPADE language, which gradually finds the optimal number of threads for stateless processing with maximum throughput. Spark Streaming [12] parallelizes streaming queries by running them on the Spark distributed dataflow framework using micro-batching. With micro-batching, the streaming computation is executed as a series of short-running Spark jobs. Each Spark job outputs incremental results based on the most recent input data. A limitation of such an execution model is that it makes it challenging to support arbitrary window semantics for continuous queries and in particular sliding windows. The Stratosphere project [13] has developed a distributed dataflow framework that can execute data-parallel batch and streaming processing jobs on the same platform. Computation is described as dataflow graphs, which are optimized using existing database techniques. The results of the Stratosphere project were made available

through the open-source Apache Flink [14] platform now exploited by the German startup Data Artisans and the only competitor to Apache Spark made by the American startup DataBricks. All the above platforms assume that stream processing operators are stateless, which simplifies scalability and failure recovery. However, this means that streaming queries cannot express complex analytic tasks such as data mining and machine learning algorithms that incrementally refine a model.

To address this problem, the fourth generation of stream processing engines adopt a stateful stream processing model. Platforms such as Apache Samza [15] and Naiad [16] execute streaming operators in a data-parallel fashion while allowing operators to have access to mutable in memory state. For example, the state of a continuous query can be a machine learning model that is trained with new incoming data. These stateful stream processing platforms therefore support the execution of analytical applications that maintain historic data while continuously processing new data. Some of these fourth-generation streaming engines rely on the concept of Stateful Dataflow Graphs (SDGs) [17]. An SDG contains vertices that are data-parallel stream processing operators with arbitrary amounts of mutable in-memory state, and edges that represent the stream. SDGs can be executed in a pipelined fashion so to have a low processing latency. All operators are assigned to machines in the cluster and the parallelization level for each operator is automatically decided by the system.

Today, some of the top tools often used for real-time data streaming processing are Apache Storm, Apache Spark, Apache Flink, Amazon Kinesis [18], Apache Samza and IBM InfoSphere Streams. Below is an extended analysis of these tools.

### 3.1.1 Apache Storm

Built by Twitter, Apache Storm [10] specifically aims at the transformation of data streams and it is useful for ETL, online machine learning, continuous computation, and many other things. The foremost capability of Apache Storm is faster data processing that can carry out processes at the nodes with faster data processing than other tools do, combined with very low latency. However, Apache Storm is known to have a few drawbacks such as that it is only suited for data which are ingested as one entity and it cannot guarantee that the data will be processed only once, and thus may compromise reliability..

### 3.1.2 Apache Spark

Spark [12] is an general-purpose distributed cluster computing framework. It is known for its in-memory processing capabilities where its engine conducts analytics, ETL, machine learning, and graph processing on data in motion or at rest. It is not actually a real-time system, but it processes in the micro-batches at a defined interval. It offers high-level APIs for different programming languages and when it has some latency, which eliminates some real-time analytics use cases, it makes sure that the data is processed in a trustworthy manner.

### 3.1.3 Apache Flink

Flink [14] is based on the concept of streams and transformations and is like a hybrid between the Spark and Storm, providing frameworks for both streaming and batch processing, approaching batches as data streams with finite boundaries. This allows Flink to be low latent while also exhibiting the data fault tolerance of Spark. It can also have several user-configurable windowing and redundant settings in order to support user configuration. In addition, Flink also implements Apache Beam, which is the contribution of Google to enable real-time processing. Some other benefits that Flink offers are:

- Stream-first approach which offers low latency and high throughput.
- Real entry-by-entry processing.

- Does not require manual optimization and adjustment to data it processes.
- Dynamically analyses and optimizes tasks.
- It is a continuous stream processing operator.

However, Flink has some scaling limitations.

### 3.1.4 Amazon Kinesis

Amazon Kinesis [18] is a robust managed service that is easy to set up and maintain which allows streaming Big Data with Amazon Web Services (AWS). In fact, it is a cloud-based service that has the capability to do real-time data streaming and processing. It helps to analyse the real-time data, scaling according to the different requirements. One of the most crucial traits of Amazon Kinesis is flexibility that helps enterprises start initially with basic reports and insights on data. Subsequently, with the growth of demand, Kinesis can help in the deployment of machine learning algorithms to support in-depth analysis. However, the fact that it is a commercial cloud service, priced per hour, makes some enterprises to conclude to other free and open-source solutions.

### 3.1.5 Apache Samza

Apache Samza [15] is considered one of the best real-time stream processing frameworks. It is designed to match with the unique architecture of Kafka, another real-time data streaming tool, and it guarantees any kind of fault tolerance. Apart from just fault tolerance, it can also work against buffering and state storage. Samza also has great scalability and is distributed on all levels, managing things like snapshotting and restoration of the stream processor's rate. Nevertheless, Samza does not offer any reliability and recovery accuracy.

### 3.1.6 IBM InfoSphere Streams

InfoSphere [7] Streams is designed to uncover meaningful patterns from information in motion (data flows) during a window of minutes to hours, as it is a full CEP system, as opposed to the aforementioned tools. The platform provides business value by supporting low-latency insight and better outcomes for time-sensitive applications, such as fraud detection or network management. InfoSphere Streams also can fuse streams, enabling to derive new insights from multiple streams. The main design goals of InfoSphere Streams are to:

- Respond quickly to events and changing business conditions and requirements.
- Support continuous analysis of data at rates that are orders of magnitude greater than existing systems.
- Adapt rapidly to changing data forms and types.
- Manage high availability, heterogeneity, and distribution for the new stream paradigm.
- Provide security and information confidentiality for shared information.



### 3.1.7 Brief Comparison of Analysed Data Streaming Tools

The following table presents a brief comparison between the tools previously analysed in terms of their capabilities and how they work. In particular, the comparison fields are the execution model that they follow, the workload which is if they are CPU or memory intensive, where the latter implies that the main performance bottleneck at higher load conditions will be due to lack of memory. Also, fault-tolerance, latency and throughput are some other fields that have been used for the comparison, along with their application domain or areas.

Table 2: Comparison of Data Streaming Tools

Tool	Execution Model	Workload	Fault tolerance	Latency	Throughput	Application
<b>Apache Storm</b>	Streaming	CPU / memory intensive	Replication, checkpoint, data recovery, upstream backup, record-level acknowledgment, stateless management	Very low	Low	Internet of things, streaming machine learning, multimedia analysis
<b>Apache Spark</b>	Batch, Iterative, Streaming	CPU / memory intensive	RDD based check-pointing, parallel recovery, replication	Low	High	Event detection, streaming machine learning, fog computing, interactive analysis, multimedia analysis, cluster analysis, filtering, re-processing, cache invalidation
<b>Apache Flink</b>	Streaming, Batch, Iterative, Interactive	Memory intensive	Stream replay and marker-checkpoint	Very low	High	Optimization of e-commerce search result, network / sensor monitoring and error detection, ETL for business intelligence infrastructure, machine learning
<b>Amazon Kinesis</b>	Streaming, Batch processing	Memory intensive	Checkpoint	Very low	High	Real-time dashboards / businesses / operational decisions, exceptions capturing, alerts generation, recommendations driving
<b>Apache Samza</b>	Streaming, Batch processing	Memory intensive	Checkpoint	Very low	High	Filtering, re-processing, cache invalidation
<b>IBM Infosphere Streams</b>	Streaming	Capture database workloads and replay them in a test database environment	Automatic recovery	Low	High	Space weather prediction, physiological data streams analysis, traffic management, real-time predictions, event detection, visualization

## 4 INFINITECH Enablers for SQL Operators over Streaming Data

As indicated the previous section, modern streaming processing frameworks nowadays provide the ability to correlate data *at-rest* with data coming from a streaming channel. They offer a variety of operators that enables a data analyst to apply processing methods on the stream, using either CEP built-in functions or high level data frames. In the latter case, streams are transformed into unbound virtual tables that can be consumed by SQL-alike operators, or other operators. This level of virtualization of the data, allows for streams to be expressed as tables, and thus, being correlated with static data that are also expressed in tabular format. The sources of the static data can vary from static files, to database management systems and other sources of persistent storage. All those have to implement a specific connector in order for the streaming processing framework to be able to retrieve and store data to the target source.

Although the ability for correlating static data with a streaming process is not novel, there are various barriers that prevent those frameworks from delivering real-time BI. Those limitations are usually introduced by the persistent storage elements, which are either unable to handle data ingestion in very high rates, or they can insufficiently execute data retrieval operations, due to the high latency that a *scan* operation requires. Regarding the latter case, a typical scenario can be to compare a streaming tuple with an aggregated value: For instance, the value of a finance transaction might need to be checked against the overall average of finance transactions that have taken place during the last defined period of time. However, this operation requires firstly a *scan* of a data partition, which typically is costly. In order to overcome this, usually there are two approaches: The first one is to *cache* this value, with the drawback that the average value is not consistent, which is not acceptable in use cases coming from the finance sectors. The second approach is to create a virtual table with the target dataset and apply the aggregation in memory. This has two benefits: It is much more effective, as all calculations take place in memory, which is less time consuming, and, continuous updates and data modifications can be applied to the common shared dataset. However, restrictions on the overall size of the memory of the dataset and insurance of the transaction semantics are a significant drawback.

Regarding operational workloads, modified data arriving in high rates must be stored in a persistent volume. Traditional database management systems usually are incapable to handle these loads, due to the enforcement of transactions. As the rates goes high, the transactional management subsystem of the database needs to scale out, in order to serve these loads. However, the distribution of transactions is hard to be achieved, as the traditional implementations make use of the *two-phase-locking* protocol, which cannot be distributed by design. To make things worse, operational workloads cannot be combined with analytical processing, as the one blocks the other. To overcome this problem, data ingestion is targeting operational datastores, while ETLs are used to periodically move data to a data warehouse. By doing this, it is possible for a streaming processing framework to ingest data to one datastore element and use the data warehouse for analytical operations. As data in the warehouse are added periodically by the execution of the ETL, the data can be considered as non-modified, can be *cached* in memory of the streaming engine. This leads to have a *near* real-time BI, as the processing takes into account a snapshot of the dataset that has been retrieved in the last invocation of the ETL. This is often not enough in modern cases coming from the finance sector, where real-time identification of potential opportunities or mal-detections is the requirement.

As it can be concluded, it is hard for a data analyst to make use of a streaming processing framework in order to correlate static and streaming data for real-time analytics. Towards this direction, the INFINITECH platform provides enablers that can be used in order to overcome those obstacles. As a result, instead of using *virtual* or *materialized views* over a dataset that allows the execution of table functions and SQL queries over correlated streaming data and data *at-rest*, while at the same time apply possible data modifications on the view, the whole architecture can be much more simplified: we can have the direct use of a data table of the data management layer. The latter can allow for the streaming framework to delegate

the requirements for data consistency to the database. The data management layer offers specific enablers that aim to overcome the obstacles introduced by the need for persistent storage.

## 4.1 Hybrid Transactional and Analytical Processing

The provision of Hybrid Transactional and Analytical Processing is at its core of the overall data management layer of INFINITECH. As it has been reported in the corresponding deliverables of task T3.1 (“Framework for Seamless Data Management and HTAP”), the purpose of this enabler is twofold: Firstly, it allows for the combined execution of operational and analytical workloads, which is crucial when there is the need for real-time business intelligence. This removes the necessity for moving data from the transactional data store to a data warehouse. It is based on the removal of all data locks that are needed by traditional implementations to enforce data consistency on transactions that are being executed in parallel. The lack of data locks allows for an analytical operation to perform a *scan* over the whole dataset (which is typically the case when we need to calculate an aggregated value) without being blocked by data modification operations that put the *locks*. That way, the data analyst or application developer does not have to create and maintain in memory specific *virtual* or *materialized views*, which are used by the streaming frameworks to share this information across streaming sessions, and delegates this responsibility to the lower layer that has been designed to serve this. In addition, the need to maintain the state across sessions and share it across the different deployment nodes is removed, along with the restriction for the size of the *view* due to memory limitations of the deployment.

Complementary to the above is the ability of the INFINITECH data management layer to handle very high rates of data ingestion. Due to its highly scalable transactional management system, it can be scaled out linearly to hundreds of nodes. As a result, it can serve hundreds of thousands of transactions per second, without being a bottleneck. This innovation of the platform allows the ingestion of data to be handled on the runtime, avoiding the need to push the incoming data for a temporal persistent and fault-tolerant queue (e.g. Apache Kafka). The approach that involves a data queue demands a consumer process that periodically gets data from the queue and puts them to the persistent storage in a batch. As a result, data are being ingested periodically in micro-batches, and this design downgrades the real-time processing to *near* real-time.

## 4.2 Online Aggregations

Another obstacle that appears when correlating streaming data with data *at-rest* is the need to combine the value of a tuple coming from the streaming channel with an aggregated value of the data already stored. Requesting the min/max/average value of a dataset to be used in a later comparison firstly requires the *scan* of the data table in order to retrieve the aggregated value. Having the dataset in a persistent storage will require lots of I/O operations to that volume, which is time consuming with a significant latency. To overcome this problem, data are being cached into memory where this value has been pre-calculated in advance. The drawback of this technique is in the case of datasets that are being modified frequently; we lose the data consistency, as the aggregated value will be outdated. *Materialized views* are usually used to deal with this requirement; however, each aggregated operation has to be calculated again each time the dataset is being modified. Even if this calculation takes place in memory, introducing serious barriers regarding the overall size of the dataset, it is computational intense. INFINITECH provides the ability to execute *online aggregations*, which means the value can be retrieved online at runtime. In contrast with the need for a *scan* operation and the calculation of the aggregated value by calculating all involved records, INFINITECH’s data management layer maintains an additional record for each involved value. The calculation is being performed on the fly, as a new record arrives. Instead of having to check the value of each record of the *scan*, the platform relies on *delta operators* that have been implemented in the scope of task T5.3 (“Declarative Real-Time Data Analytics”). As a result, the value has been pre-calculated

and the complexity for data retrieval is only  $O(1)$ , which is the minimum we can get. Furthermore, being already integrated with the transactional management component of the platform, it is ensured that the value is consistent in terms of transactional semantics. This will remove the necessity for the streaming processing framework to maintain such expensive in terms of resource usage and time consuming *views*, and downgrades this to the lower layer. More details regarding the implementations of the *online aggregators* can be found in the corresponding deliverables of T5.3.

### 4.3 Polyglot Capabilities

We can consider a third obstacle, the necessity to correlate static data coming from different data sources. Most of the streaming processing frameworks implement a variety of *join* operators that can be used to get data from various sources and correlate the results between them and among streaming channels. However, a *join* operator usually requires to get into memory a significant amount of data that needs to be used for the outer operator of the operation itself. INFINITECH's polyglot component can be used instead, so that the data analyst and application developer can write a simple *select-from* statement to get the corresponding result set. By doing this, she pushes down to the data management layer of INFINITECH the execution of this join, removing the need from the streaming processing to maintain all data in memory. As explained in the corresponding deliverables of T3.2 ("Polyglot Persistence over BigData, IoT and Open Data Sources"), the polyglot component can receive a query written in a common language and execute this in the various target datastores. As a result, the retrieval of the static data becomes more transparent, as this is being delegated to another component, letting the streaming engine to only correlate the streaming data with the result.

### 4.4 Incremental Analytics

Based on the advancements on the work that has been carried out under the scope of task T5.2 ("Incremental and Parallel Data Analytics"), a forth enabler that has been developed in INFINITECH can be now exploited by our streaming processing framework. This is the provision of incremental analytics that has extended the base functionalities of INFINISTORE. Using the incremental analytics, the data user or application developer can submit a continuous or un-bounded query statement to the datastore that will be validated for a given duration of time (or for infinite) against the dataset that changes its state continuously. As data is being ingested to the datastore, all incoming tuples will be checked whether or not they validate the submitted query, and if yes, an updated row (or number of rows) will be returned. This mechanism is now documented in D5.3 ("Library of Parallelized Incremental Analytics – III"). In summary, when the data user or application developer wants to submit an un-bounded query, he or she needs to open a database connection to submit the query, and he or she will be returned an *iterator* object to gather the results. In the case of un-bounded queries, this iterator never stops, but instead, waits for new records to become available from the storage engine of INFINISTORE. With the use of incremental analytics, the data user can submit aggregation operators (i.e. the average of financial transaction movements per credit card per time period), and will get updates of this results each time new records are available and ingested in the datastore. This means that in practice, the open *iterator* creates a stream of data that can be consumed by the data users. Having this, it will allow the streaming processing framework to handle the results of the incremental analytics as an external data stream using materialized views defined by the data user. In fact, the INFINISTORE, as the data management layer of INFINITECH, can ensure data stream with consistent data, already pre-processed and aggregated with no latency, as all these operations takes place in the datastore itself. Therefore, the result is that the data analyst can push down all this complex processing down to the INFINISTORE, and let the latter create a streaming data to be handled by the streaming processing framework. He or she write SQL statements using relational algebraic operations, to be processed in the datastore, and consume or combine the output as a stream. Having the incremental

analytics embedded into the INFINISTORE and accessed via its direct API, then it can be integrated with the streaming processing framework, the same way as the other three enablers are.

## 5 INFINITECH streaming engine overview and design of operators

After performing an intensive state-of-the-art analysis of existing solutions regarding frameworks for streaming processing, which can be found in Section 3, we decided to use the Apache Flink as the basis for the INFINITECH unified query processing framework, as it offers a variety of characteristics and functionalities that can be useful for the platform. It provides two different APIs for correlating streaming data with data *at-rest* which transforms streams and external persistent data into tabular formats that can be used by the various data operators. The provided Table API and SQL API can be used for static data and can be used in conjunction with the *DataStream* and *DataSet* interfaces.

Apache Flink provides two manners for processing: a language-integrated query API and the ability to direct execute SQL statements. The former provides a set of available methods that can be invoked and can compose a pipeline of relational operations in an intuitive way and returns an equivalent result compared to a normal SQL execution. It supports common SQL operators such as selections, projections, filters, aggregations etc. The latter manner requires the compilation of the SQL statement to an execution plan that will be applied. The compilation makes use of Apache Calcite [21].

There is one significant difference however between those two ways for accessing static data: Using SQL operators, the whole dataset needs to be available to the streaming engine, where the latter applies the query plan in memory and returning back the results. The only exception is for SQL-compatible data management systems, where the whole query can be pushed down to the source via a Java DataBase Connection (JDBC) connection. In fact, in those cases, it is the database itself that takes care of the execution of the statement and returns back the result set, which will be further transformed to a tabular format, thus initializing a *Table* instance of the corresponding API. In all other cases, the dataset has to be fetched first from the source, and the execution plan needs to be handled by Flink in memory. An alternative approach is the implementation of specific *connectors* that can be used by the streaming engine using the language-integrated query API. As mentioned before, the query API provides various relational operations such as selections, projects etc, and the *connector* implements those operators for data access. As a result, the dataset does not have to be loaded in memory. Instead, those operations are being executed in the target datastore which filters out records and returns the results.

The INFINITECH data management layer is SQL-compatible and implements the JDBC specification. However, this requires the invocation of the query engine that introduces an inherit overhead due to its footprint, while on the same time, does not take the fully advantage of the recent enablers of INFINISTORE developed under the scope of this project during its last phase. Due to this, we now provide an INFINISTORE Flink connector, which implements all operations supported by Flink for unified stream and batch processing. The benefit is twofold: firstly, it allows for direct access to the storage engine of the platform, bypassing the footprint introduced by the query engine, and as a result, can support data ingestion in even higher rates. What is more, the direct API of the data storage of INFINITECH has been designed to support the distributed execution of aggregated operations. As a result, these types of operations can be pushed down to the storage for efficient data retrieval. The following sections provide information about the initial design of integration of the streaming engine of Flink with the INFINITECH data management layer via those operators.

### 5.1 Basic Concepts

The Table API of Apache Flink has as its central concept the *Table* which serves as the input and output of operations: an operator that performs a project will take as input an instance of a *Table* and will return the projected result in another instance of a *Table*. Tables can be either permanent or temporary. The former

allows to be visible across several sessions that might be span across different nodes, while the latter is only visible during the lifecycle of a single session.

In order to create a table, there are two different ways: one is to use the native API that allows you to use the language-integrated query, which composes the operators by using native language, or by using an SQL statement and pass the query string to the framework. The following code snippet shows how to construct a table with the native API.

```
// create a Table
tableEnv.connect(...).createTemporaryTable("Orders");

// scan registered Orders table
Table orders = tableEnv.from("Orders");

// compute revenue for all customers from France
Table revenue = orders
    .filter($"cCountry").isEqual("FRANCE")
    .groupBy($"cID", $"cName")
    .select($"cID", $"cName", $"revenue").sum().as("revSum");
```

While this code snippet produces an equivalent result, using an SQL statement:

```
// create a Table
tableEnv.connect(...).createTemporaryTable("Orders");

// scan registered Orders table
Table orders = tableEnv.from("Orders");

// compute revenue for all customers from France
Table revenue = tableEnv.sqlQuery(
    "SELECT cID, cName, SUM(revenue) AS revSum " +
    "FROM Orders " +
    "WHERE cCountry = 'FRANCE' " +
    "GROUP BY cID, cName"
);
```

The difference between these two code snippets is the way they retrieve data from the underlying datastore. In the second example, there is an SQL string that will get all orders from the country whose name is *FRANCE*, and will return the overall revenue of all customers living in that country. To do so, it will need to use the *FROM* clause in order to do a selection over the *ORDERS* data table, then apply a filter condition via the *WHERE* clause and the *GROUP BY* clause to group the summary of the values over those columns. Finally, it will project the two columns in the *GROUP BY* and will apply the aggregation operator over the *revenue* column. This query can be pushed down via the *JDBC* in cases Flink is integrated with an SQL compatible data source. Otherwise, it will grab all data from the table *ORDERS* first, and then it will apply this query over the dataset that has been fetched in memory.

In the other code snippet however, the same query is expressed via the native API. We can see the involved operators are being constructed step-by-step. Given that, a *filter* will be applied on the specific column over the table that have been defined, which will get data from the table *ORDERS*. Then, the group by method will be invoked whose result will be projected by the *select* method, which also applies the aggregation operation.

Apart from reading data, the API provides the ability to manipulate and persist data to a persistent storage. The following code snippet provides an example.

```
// create an output Table
final Schema schema = new Schema()
    .field("a", DataTypes.INT())
    .field("b", DataTypes.STRING())
    .field("c", DataTypes.BIGINT());

tableEnv.connect(new FileSystem().path("/path/to/file"))
    .withFormat(new Csv().fieldDelimiter('|').deriveSchema())
    .withSchema(schema)
    .createTemporaryTable("CsvSinkTable");

// do something and get the result
Table result = ...

// emit the result Table to the registered TableSink
result.executeInsert("CsvSinkTable")
```

This code creates a temporary table called `CsvSinkTable` that is mapped to a csv file in the storage and has the schema that has been defined at the first lines of the code. After doing a process, the data analyst retrieves the data and puts them into an instance of the *Table*, and then invokes the `executeInsert` method to actually store data into the csv file.

## 5.2 Stream Correlation with Data At-Rest

Apache Flink provides two APIs that allows the manipulation of streaming data, which are the *DataStream* and *DataSet* APIs. *Table* API and SQL queries can be easily integrated with and embedded into *DataStream* and *DataSet* programs. As a result, the data analyst can write a query to retrieve data from an external data table that is stored in a relational database management system and do a pre-processing: apply some filters, aggregate data that are grouped by a number of columns and project specific columns to the temporary *table*. The data stored in the *table* can be further processed with either the two of the *DataStream* or *DataSet* APIs. The same can happen vice versa: it is possible for a *DataStream* or *DataSet* program to be used as an operand in an operator that is part of the *Table* API.

Being able to transform those two APIs gives the ability for the streaming engine to correlate data of those two different types: streaming data with data *at-rest*. As data stored in the *table* can be further processed by the streaming APIs, it allows data coming from a stream to make use of static information that can be retrieved by query statements over a persistent data source. Having said that, we can retrieve the average value of the finance transactions of a user during the past week, by executing an analytical query to the target database, and retrieve this result via a *Table*. This value can later on be used by the streaming APIs to check if the value of a current finance transaction is bigger than the amount of money that this customer is usually performing, that might trigger an alert.

In the same sense, an operator of the *Table* interface might be able to insert data to a persistent storage. The ability of the *Table* to consume data of the streaming APIs allows for the direct ingestion of data streams into the storage layer. As it has been already mentioned, the insert operation usually puts data into a data queue, and a consumer process periodically sends micro batches to the target datastore. In INFINITECH, we have designed our insert operator to directly ingest data to the storage engine of the platform, taken advantage of its ultra-scalable transactional manager that allows to server operational workloads in very high rates.

Moreover, in order to correlate stream and batch data, it is not enough to simply be able to transform the different types of APIs, but also providing a framework and operators that can be applicable to both types of data. Their main differences are that batch data are bound, while streaming is usually unbound, batch data pre-exist while streaming data continuously fills the query and batch data produces static results,



while streaming data continuously change the result as the stream goes through the operator. In order to overcome those differences, there has been introduced the concept of *virtual views*. All input of the operators implements such a *virtual view* so that the execution of the operator can be transparent from the implementation of the view. More precisely, in order to deal with streaming data, there has been proposed the *Materialized view* or *Dynamic Table*. The latter caches the result of the query such that the query does not need to be evaluated each time the view is being accessed. However, the data in the view can be outdated when a data modification operator arrives into the stream. In order to overcome this, different techniques can be applied that updates the materialized view, by listening to changes by data modification operators of the stream. Dynamic tables are changing over time in contrast to the static tables that represent batch data. Due to this, queries targeting streaming data are often called *Continuous Queries*, which never terminate and produce those dynamic tables as the result. This means that those queries continuously update their result in order to reflect the changes on its dynamic input tables.

Taking into account that the maintenance of the updates coming from the stream in the dynamic table must be done in memory, this concept comes with several restrictions, mainly regarding computational and memory usage. Continuous queries are evaluated on unbounded streams and are often supposed to run for weeks or months. As a result, the total amount of data that a continuous query processes can be very large. Similarly, other queries require re-computing and updating a large fraction of the emitted result rows even if only a single input record is added or updated.

Dynamic tables are the core concept of Flink’s Table API and SQL support for streaming data. In contrast to the static tables that represent batch data, dynamic tables are changing over time. They can be queried like static batch tables. Querying dynamic tables yields a Continuous Query. A continuous query never terminates and produces a dynamic table as result. The query continuously updates its (dynamic) result table to reflect the changes on its (dynamic) input tables. Essentially, a continuous query on a dynamic table is very similar to a query that defines a materialized view.

## 5.3 INFINITECH Operators for the Streaming Engine

As it has been described in the previous subsections, Apache Flink will be used as the basis for the Unified Query Processing Framework of INFINITECH that will allow the correlation of streaming data with *at-rest*. Even if natively this framework provides the tools and APIs to mix streaming with batch processing, they come with certain limitations so that the overall solution cannot be used for real-time business intelligence. To overcome these limitations, we decided to move all *materialized views* to be handled by the data management layer. This will imply that all data modification operators coming from the stream will be targeting the central data repository of the platform. Data stored in its data tables can be shared across different Flink sessions, so there is no need for the latter to create and bind temporary tables to sessions. Our design relies on the technological enablers of INFINITECH, as briefly described in Section 0, while more analytical details can be found at the relevant deliverables of the corresponding tasks of the project. In order to support data ingestion in very high rates, it was considered that accessing the storage engine of the platform will increase the overall throughput, as the latency will become lesser, due to the fact that a modification operation will avoid the performance overhead introduced by the footprint of the query engine. For this reason, an INFINITECH *connector* with Flink has to be provided that implements a group of specific operators. The delivery of those operators is the main objective of Task 3.3, and an initial list, along with some implementation details can be found in this subsection.

### Create/Alter Table Schema

- **AddColumns:** Performs a field add operation. It will throw an exception if the added fields already exist.

- **AddOrReplaceColumns:** Performs a field add operation. Existing fields will be replaced if add columns name is the same as the existing column name. Moreover, if the added fields have duplicate field name, then the last one is used.
- **DropColumns:** Performs a field drop operation. The field expressions should be field reference expressions, and only existing fields can be dropped.
- **RenameColumns:** Performs a field rename operation. The field expressions should be alias expressions, and only the existing fields can be renamed.

A code snippet on how to invoke these methods from Flink can be found as follows:

```
Table orders = tableEnv.from("Orders");
//AddColumns
result = orders.addColumn(concat($"c", "sunny"));
//AddOrReplaceColumns
result = orders.addColumn(concat($"c", "sunny").as("desc"));
//DropColumns
result = orders.dropColumns($"b", $"c");
//RenameColumns
result = orders.renameColumns($"b".as("b2"), $"c".as("c2"));
```

This will require the connector to drop the table and to recreate with the corresponding schema, as defined in the Flink client. The following code snippet demonstrates how this is implemented in the connector, according to the type of mode to append.

```
Settings settings = relationSettings.buildSessionSettings();
try(Session session= SessionFactory.newSession(relationSettings.getUrl(),settings)) {
    String table = relationSettings.getTable();
    boolean exists = session.database().tableExists(table);
    if(exists) {
        switch (mode) {
            case ErrorIfExists:
                throw new IllegalArgumentException(String.format("Table %s already exists", table));
            case Overwrite:
                session.database().dropTable(table);
                createTable(session,table,keyFields,data.schema());
                break;
            case Ignore:
                return;
        }
    }
    else{
        createTable(session,table,keyFields, data.schema());
    }
}
catch (RuntimeException e){
    throw e;
}
catch (Exception e){
    throw new LeanxcaleRuntimeException(e);
}
```

### Scan, Projection, and Filter

- **From:** Similar to the FROM clause in a SQL query. Performs a scan of a registered table.
- **Values:** Similar to the VALUES clause in a SQL query. Produces an inline table out of the provided rows.
- **Select:** Similar to a SQL SELECT statement. Performs a projection operation.
- **As:** Renames fields.
- **Where / Filter:** Similar to a SQL WHERE clause. Filters out rows that do not pass the filter predicate.

A code snippet on how to invoke these methods from Flink can be found as follows:

```
Table orders = tableEnv.from("Orders");
Table result = orders.select($"a", $"c").as("d")
    .as("x, y, z, t")
    .where($"b").isEqual("red");
```

This will require the connector to implement all corresponding filter methods that can be found in Flink. To handle these cases more effectively, a *FilterTranslator* interface have been defined in the connector, which defined a *filter* method, that each of the operations implements accordingly. The *FilterTranslator* can be found in the following code snippet:

```
public interface FilterTranslator<T extends org.apache.flink.sql.sources.Filter>{
    Filter translate(T filter, TableModel tableModel);
}
```

An implementation of this method for the *isEqual* filter method of the example can be found in this code snippet:

```
@Override
public Filter translate(EqualTo filter, TableModel tableModel){
    String field = filter.attribute();
    Type type = tableModel.getFieldType(field);

    switch (type){
        case SHORT:
        case INT:
        case LONG: return Filters.eq(field, ((Number)value).intValue());
        case FLOAT:
        case DOUBLE: return Filters.eq(field, ((Number)value).doubleValue());
        case TIMESTAMP: return Filters.eq(Expressions.field(field), Constants.timestamp((Date)value));
        case DATE: return Filters.eq(Expressions.field(field), Constants.date((java.sql.Date)value));
        case TIME: return Filters.eq(Expressions.field(field), Constants.time((Time)value));
        case STRING: return Filters.eq(field, (String)value);
        case BOOLEAN: return Filters.eq(Expressions.field(field), Constants.bool((Boolean)value));
        default: throw new IllegalArgumentException(String.format("Type not %s supported", type));
    }
}
```

### Aggregations

- **GroupBy Aggregation:** Similar to a SQL GROUP BY clause. Groups the rows on the grouping keys with a following running aggregation operator to aggregate rows group-wise

The following code snippet shows how to invoke this method via Flink

```
Table orders = tableEnv.from("Orders");
```

```
Table result = orders.groupBy($"a")
    .select($"a", "b")
        .sum()
        .as("d");
```

### Distinct

- **Distinct:** Similar to a SQL DISTINCT clause. Returns records with distinct value combinations.

The following code snippet shows how to invoke this method via Flink

```
Table orders = tableEnv.from("Orders");
Table result = orders.distinct();
```

### Joins

Apache Flink supports various types of *join* operations, such as inner, outer or interval joins. This operator cannot be pushed down directly to the data storage of the INFINITECH platform, as its API does not support joins between tables. As a result, the user needs to use the JDBC driver to retrieve data from the data management player, and write a standard SQL statement, which will be executed by the query engine of the platform.

### Order By

- **Order By:** Similar to a SQL ORDER BY clause. Returns records globally sorted across all parallel partitions.
- **Offset & Fetch:** Similar to the SQL OFFSET and FETCH clauses. Offset and Fetch limit the number of records returned from a sorted result. Offset and Fetch are technically part of the Order By operator and thus must be preceded by it.

A code snippet on how to invoke these methods from Flink can be found as follows:

```
Table in = tableEnv.fromDataSet(ds, "a, b, c");
in.orderBy($"a").asc()
    .offset(10)
    .fetch(5);
```

### Insertions

- **Insert Into:** Similar to the `INSERT INTO` clause in a SQL query, the method performs an insertion into a registered output table. The `executeInsert()` method will immediately submit a Flink job which execute the insert operation.

A code snippet on how to invoke this method from Flink can be found as follows:

```
Table orders = tableEnv.from("Orders");
orders.executeInsert("OutOrders");
```

The *outOrders* table contains a list of *rows* that will be inserted into the datastore, via the connector. The implementation of this operator will execute the following code in order to insert the tuples to the data storage:

### D3.8 – Data Streaming and Data at Rest Queries Integration - III

```
Settings settings = relationSettings.buildSessionSettings();
try(Session session = SessionFactory.newSession(relationSettings.getUrl(), settings)){
    int totalCount = 0;
    try {
        Table table = session.database().getTable(relationSettings.getTable());
        int commitCount = 0;
        while (iterator.hasNext()) {
            Tuple tuple = table.createTuple();
            Row row = iterator.next();
            for (int i = 0; i < fieldsByPos.length; i++) {
                tuple.put(fieldsByPos[i], row.apply(i));
            }
            table.insert(tuple);
            totalCount++;
            if(++commitCount>=commitRows){
                session.commit();
                commitCount = 0;
            }
        }
        if(commitCount>0) {
            session.commit();
        }
    }
    catch (Exception e){
        log.warn("Exception writing row {}, rollbacking transaction",totalCount);

        try{
            session.rollback();
        }
        catch (Exception e1){
            log.error("Exception doing rollback {}",e1.getMessage(), e1);
        }
        throw e;
    }
}
catch (RuntimeException e){
    throw e;
}
catch (Exception e){
    throw new LeanxcaleRuntimeException(e);
}
```

## 6 The INFINISTORE Flink Connector

After an exhaustive analysis of the state-of-the-art technologies used for unified query processing of both streaming data and data *at-rest*, and after deciding to rely on the Apache Flink streaming processing framework, during the first phase of the project, we provided a distribution of Flink that relies on the JDBC connections of the INFINISTORE for accessing data from the datastore, and delegates to Flink handle the processing. In parallel, we analyzed what Flink can offer and designed our custom implementation of the Flink Operator for the INFINISTORE, based on the interfaces that Flink provides to external data providers. During the last phase of the task T3.3 that is documented in this report, we implemented this connector, whose details will be described in this section. We have implemented both the *sink* (from Flink to the datastore) and *source* (from the datastore to Flink) mechanisms of the connector, that taking advantage the underlying technologies developed in the project, will:

- allow Flink to store data to the INFINISTORE coming from a streaming with very high velocity, taking advantage of the high data ingestion capabilities of the datastore itself, as explained in D3.3 (“Hybrid Transactional/ Analytics Processing for Finance and Insurance Applications – III”)
- retrieve pre-processed aggregated data from the INFINISTORE with minimum latency, avoiding the need to either scan the whole data table in the datastore layer, or cache the table in memory and either lose data consistency or run out of memory, using the advancements of the Online Aggregates, documented in D5.6 (“Framework for Declarative and Configurable Analytics – III”)
- retrieve data coming from the INFINISTORE as an un-bound data stream, using the advancements of the Incremental Analytics, documented in D5.3 (“Library of Parallelized Incremental Analytics – III”).

We have divided this section as follows. First, we will give a general overview of the core of our implementation, and then we will dive into the details for the *sink* implementation and then for the *source* implementation. Later, we will describe how we implanted the additional *abilities* supported by the Flink that allows our implementation to further process data much more efficiently, allowing the Flink query engine to push operations down to the datastore. Finally, we will provide a demonstrator of its use.

### 6.1 The INFINISTORE Flink Connector: The core

Apache Flink allows its users to define their materialized views, and explicitly require the use of a custom connector to access an external data management system, providing its specific configuration parameters. An example of such a definition can be found in the following code snippet:

```
EnvironmentSettings settings = EnvironmentSettings.newInstance().build();
TableEnvironment tEnv = TableEnvironment.create(settings);

tEnv.executeSql("CREATE TABLE SENTIMENTS (\n" +
    "    id_str VARCHAR,\n" +
    "    sentiment VARCHAR,\n" +
    "    sentiment_score DOUBLE,\n" +
    "    created_at TIMESTAMP(3)" +
    ") WITH (\n" +
    "    'connector' = 'kivi',\n" +
    "    'connection-url' = 'lx://datastore:9876/SARGA@APP;KVPROXY=datastore!9800',\n" +
    "    'bounded' = 'true',\n" +
    "    'table-name' = 'TWEET_FINANCE'\n" +
    ")");
```

We will rely on this example in the following subsections to demonstrate the use of our connector. Here, the user defines a materialized view for a dynamic table called *SENTIMENTS* with 4 fields (*id\_str*, *sentiment*, *sentiment\_score* and a *timestamp*) and a list of configuration parameters. This will make use of a static table stored in the INFINISTORE, where we put the results of a sentiment analysis over a stream of data coming from a Tweeter, related with trends in the finance sector.

This materialized view can be now be used by the Flink query engine to both store (using the *sink connector*) and retrieve (using the *source connector*) from the INFINISTORE. When access to this materialized view is requested, then Flink will make use of the custom *TableFactory* implementation, to provide its catalogue meta-info. Flink defines two interfaces, the *DynamicTableSourceFactory* and *DynamicTableSinkFactory* to be used for inserting or retrieving data from the external data source provider, in our case, the INFINISTORE. Therefore, as depicted in Figure 2, our *KiviTableFactory* will implement both those interfaces.

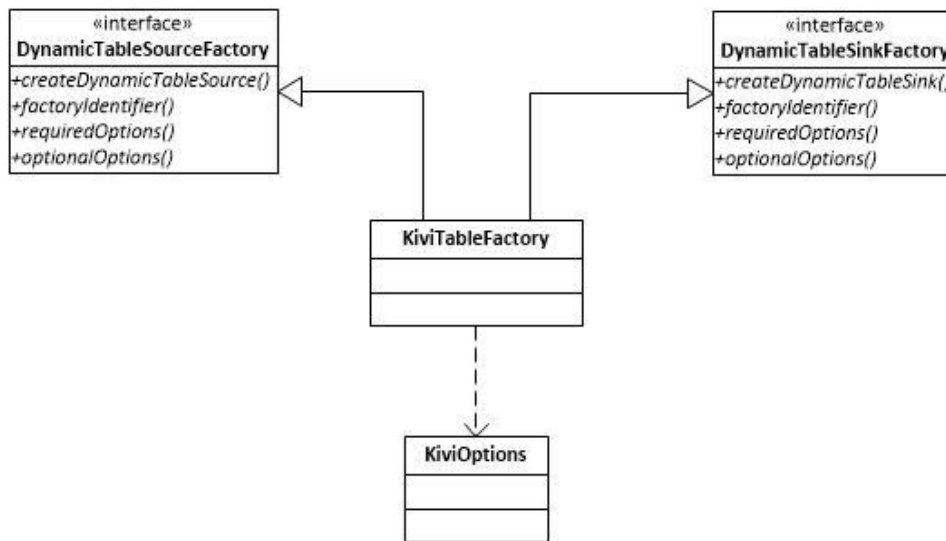


Figure 2: Flink Connector – The Core Factory

The aforementioned interfaces define a set of common methods that will return the unique identifier of our implementation, which in our case is called ‘kivi’, methods that return the options or configuration parameters that the data user has defined in his or her materialized view, and the methods to create the instances for the *sink* and *source* implementations to be later used by Flink’s internal query engine. The following code snippet illustrates the *options* or configuration parameters that our implementation expects, along with their default values (if applicable).

```

// definition of all available options
ConfigOption<String> CONNECTION_URL = ConfigOptions.key("connection-
url").stringType().noDefaultValue();
ConfigOption<String> TABLE_NAME = ConfigOptions.key("table-name").stringType().noDefaultValue();
ConfigOption<Long> POLL_FREQUENCY = ConfigOptions.key("poll-
frequency").longType().defaultValue(1000L);
ConfigOption<Integer> MAX_TRANSACTION_TIME = ConfigOptions.key("max-transaction-
time").intType().noDefaultValue();
ConfigOption<Integer> READER_QUEUE_SIZE = ConfigOptions.key("reader-queue-
size").intType().defaultValue(10000);
ConfigOption<Integer> CONNECTION_ATTEMPS = ConfigOptions.key("connection-
attempts").intType().noDefaultValue();
ConfigOption<String> STREAM_BEHAVIOUR = ConfigOptions.key("stream-
behaviour").stringType().defaultValue("infinite");
ConfigOption<Long> MAX_DURATION = ConfigOptions.key("max-duration").longType().defaultValue(0L);
ConfigOption<Integer> COMMIT_BATCH = ConfigOptions.key("commit-
batch").intType().defaultValue(10000);
ConfigOption<String> READER_FROM = ConfigOptions.key("reader-
from").stringType().defaultValue("begin");
ConfigOption<Long> OFFSET_CTS = ConfigOptions.key("offset-cts").longType().defaultValue(0L);
  
```

```
ConfigOption<Boolean> BOUNDED =
ConfigOptions.key("bounded").booleanType().defaultValue(Boolean.TRUE);
```

We can see from the code snippet parameters like *connection-url*, *table-name* etc. that was previously defined by our data user in the example we provided.

We will now go into the details of the implementation first for the *sink* part, and then for the *source*.

## 6.2 The INFINSTORE Flink Connector: The sink implementation

Having the definition of the materialized view that we illustrated in the previous example, let’s imagine that the data user now wants to insert some records into our data table, after a sentiment analysis that took place by analyzing an incoming stream of Tweets using the Apache Flink Framework. In other words, the result such an analysis will be stored in the IFINISTORE. The data user may need to write something similar to the following code snippet:

```
org.apache.flink.table.api.Table source = tEnv.from("source");
org.apache.flink.table.api.Table table = tEnv.sqlQuery("select * from source");
TableResult result = table.executeInsert("SENTIMENTS");
```

In this code, he or she will take input from a *source*, which in our case can be the result of a Flink operator that does the aforementioned sentiment analysis, and will insert the result obtained by the latter, into the materialized view called *SENTIMENTS*, that we defined in the previous subsection. When the *executeInsert* is called, Flink will examine the materialized view, will grab the *connector* attribute, which in our case is called ‘kivi’ and will try to find the *DynamicTableSinkFactory* implementation factory, whose *factoryIdentifier* method returns the value of ‘kivi’. In our case, this will be our *KiviTableFactory*. Then it invoke its *createDynamicTableSink* to get our implementation that will be used by Flink’s internal query planner and finally establish the connection to INFINSTORE and eventually store the dataset. Figure 3 depicts the class diagram of our implementation.

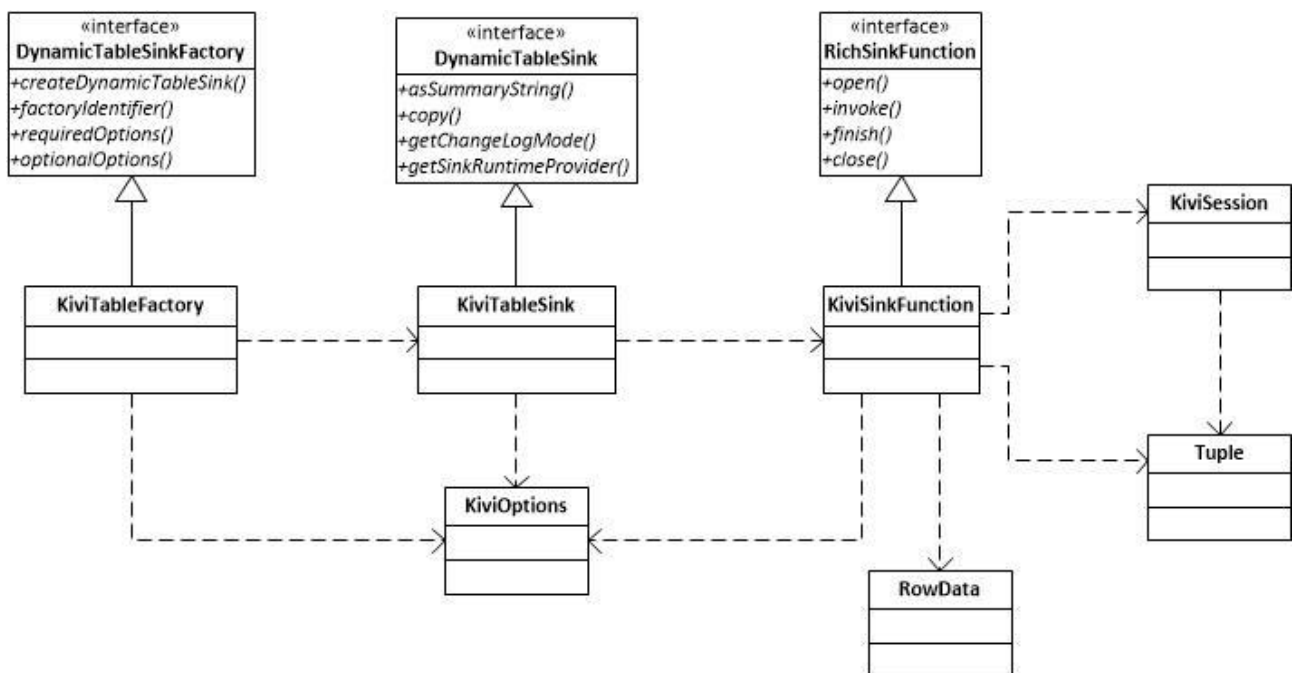


Figure 3: Flink Connector – The Sink implementation



Our *KiviTableFactory* will create a *DynamicTableSink* that will be used by the Flink’s internal planner. It implements the *DynamicTableSink* and has to implement its methods. Flink can invoke the *copy* method to create various copies of this instance to be used internally by its query optimizer and re-configure how the sink run-time implementation should behave. In the case of the *sink* implementation, there are no many options, so eventually, its *getSinkRuntimeProvider* will create the *function* or *operator* that will take the responsibility to connect to the external datastore, the INFINISTORE, and handle the logic of storing the results. Thus, an implementation of the *RichSinkFunction* will be created, which in our case is the *KiviSinkFunction*.

*KiviSinkFunction* needs to implement the *start* method that will be invoked by the Flink core engine to start the operator. Our implementation takes care of establishing the database connection to the INFINISTORE, by creating a *KiviSession* instance, and initializes all its internal state. Then, Flink will (periodically or not) call the *invoke* function, providing a list of *RowData*. This class wrappers the data coming from Flink and need to be handled by each custom implementation of the Flink connector. *KiviSinkFunction* translates these objects to *Tuples*, which is the class that represents a data row in the storage engine of the INFINISTORE database. It will communicate with the target data table, the *TWEET\_FINANCE* that was defined in the materialized view, to get the meta-information of this table. This value is passed to the runtime implementation via the *KiviOptions* that was initially created by our factory. *KiviSession* will return the schema meta-information, so that the *KiviSinkFunction* can transform the incoming *RowData* to the *Tuple* that is required by the INFINISTORE. Having the *KiviSession* data connection open, our implementation will iterate through the list of incoming data, it will invoke the *insert* operation of the opened connection and will commit after a pre-defined size of batch (retrieved by the *commit-batch* parameter in the *KiviOptions*). It is important to highlight here that no actual communication with the datastore will take place before the invocation of the *commit* method, so our implementation takes advantage of the characteristics of the INFINISTORE itself, and sends a large batch of input *tuples* (its default size is 1000) that can be handled efficiently by the datastore. That way, there is no much time spent for communicating with the datastore, and the Flink sends batches of records periodically.

At the end, the Flink core engine will invoke the *finish* operator, and our *KiviSinkFunction* implementation will finally commit all remaining *tuples* that was not committed before. And then, the invocation of the *close* method will close the opened *KiviSession* database connection, releasing all the pending open resources.

## 6.3 The INFINISTORE Flink Connector: The source implementation

As we saw how our implementation of the Flink connector works the core of the streaming processing framework for inserting data to the external datastore, in our case the INFINISTORE, now we can take a look on how it works for reading data from the datastore.

Let’s imagine that our data user wants now to read the already stored data to his or her previously defined materialized view. A first attempt will be to read everything from the target data table and use Flink’s query engine to further process data and combine them with other streaming data coming as input from a different channel. The data user may need to write something similar to the following code snippet:

```
Table testFromSourceTable = tEnv.from("SENTIMENTS");
TableResult tableResult = testFromSourceTable.execute();
```

This line of code will get everything from our materialized view called *SENTIMENTS* and will allow the data user to make use of the Flink’s Table API, as explained in a previous section. When the *execute* is called, Flink will again examine the materialized view, will grab the *connector* attribute, which in our case is called ‘kivi’ and will try to find the *DynamicTableSourceFactory* implementation factory, whose *factoryIdentifier* method returns the value of ‘kivi’. In our case, this again will be our *KiviTableFactory* that implements both interfaces. Then it invoke its *createDynamicTableSource* to get our implementation that will be used by

Flink’s internal query planner and finally establish the connection to INFINISTORE and eventually store the dataset. Figure 4 illustrates how this has been implemented in our case.

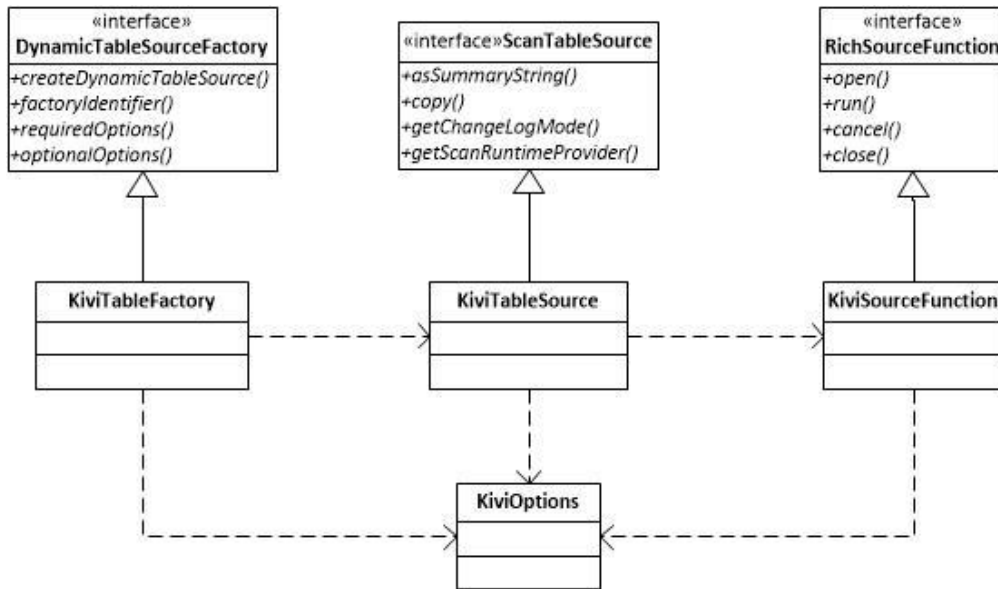


Figure 4: Flink Connector – The Source implementation

The class diagram is very similar to the one we showed for the *sink* part of our connector. The *KiviTableFactory* will create an instance of the *KiviTableSource*. The latter will be used internally by the Flink query planner to explore and examine alternative query plans that could be used for the runtime execution. The planner could create numerous variations of the *KiviTableSource*, changing or injecting additional parameters before deciding on which will make use. In this simple example, the data user wants to retrieve everything from the external datastore, so there are no other alternatives on how the planner can further improve the overall query execution and will decide to use the firstly created instance. Once this is decided, then the *getScanRuntimeProvider* method will be called, that will return an implementation of the *RichSourceFunction*, which in our case is the *KiviSourceFunction*. This will take the responsibility of the runtime execution of this *function* or operator, establish a connection to the external data source provider, which in our case is the INFINISTORE, retrieve the results and return them back to the Flink core engine to become eventually available to the data user from the Table API. The implementation of the *KiviSourceFunction* is more complicated in this case from the *KiviSinkFunction* that we documented in the previous subsection, and its details are illustrated in Figure 5.

*KiviSourceFunction* extends the Flink’s *RichSourceFunction* that defines a set of methods. As it might make use of the incremental analytics of INFINISTORE to get data from an un-bounded query, it will eventually make use of an *Iterable* object that will have to wait and block until new data can become available. Due to this, we need to put the code that gets data from the data management system in a separate thread. The implementation of our function is similar to the implementation of the *Kafka Source Connector* that we implemented under the scope of T5.2 (“Incremental and Parallel Data Analytics”) and documented in D5.3 (“Library of Parallelized Incremental Analytics – III”). Here is how it works:

After creating our *RichSourceFunction* to handle the data retrieval in run-time, the Flink core engine will invoke its *open* method. Our implementation will create an instance of the *LXReader* and pass to the latter all information it needs to establish a data connection and create the query statement it needs to submit to the INFINISTORE, using its direct API. Similarly to the *Kafka Source Connector*, the *LXReader* shares a data queue with the *RichFunction* to exchange asynchronously data retrieved from one thread to the thread that is being used by the Flink core engine. It will open the database connection using the *KiviSession* and retrieve data from the datastore as *Tuples*. As data is retrieved, the *tuple* is put in the *TupleQueue*. Once data is there, the *KiviSourceFunction* retrieves the data and transforms them to *RowData*, the class the

takes the role of a data wrapper from Flink to put the information. In order to do so, it gets the meta-info of the retrieved tuple, in order to transform the tuple to the corresponding data type expected by flink.

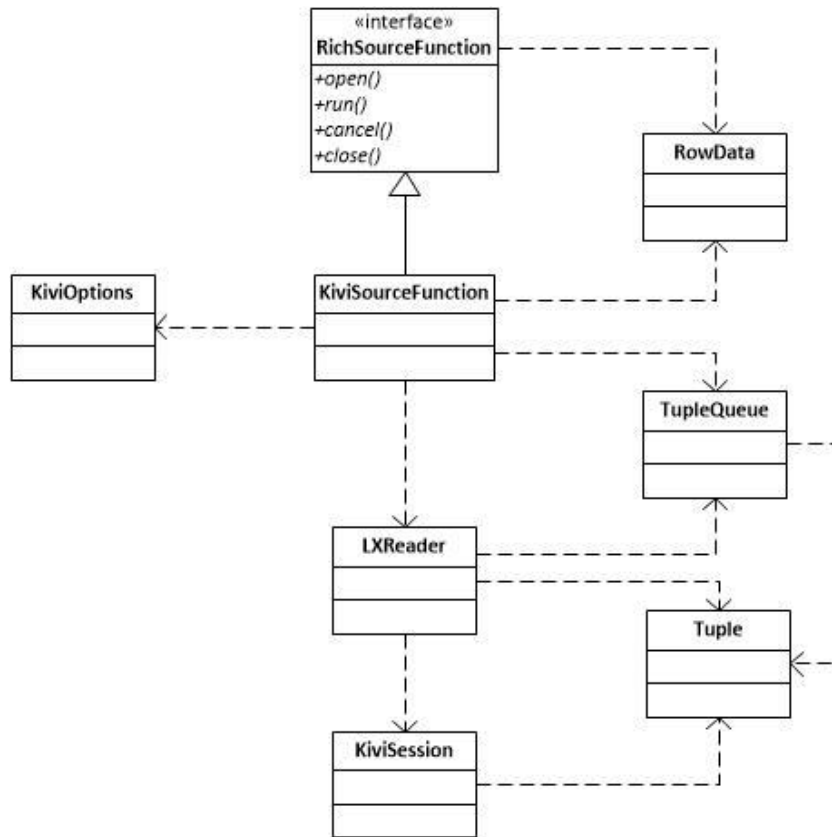


Figure 5: Flink Connector – The Source implementation: KiviSourceFunction

In the case of bounded queries, when the iterator retrieved by the *KiviSession* closes, the *RichSourceFunction's run* method will end, and Flink will gracefully close this instance. In case there is an un-bounded query, the iterator will remain open and wait for new data, due to the provision of the incremental analytics. We need to highlight here that the *RichSourceFunction* is a *Runnable object*, so its *run* method will not terminate, and will continuously make a loop and check for new values every given period of time, defined by the *poll-frequency* configuration parameter defined in the materialized view of this table.

In case Flink decides to stop the run-time execution of this operator, the *cancel* operation will be invoked, that will raise a flag for the operator to stop. This will make the *run* method to eventually finish, and then Flink will invoke the *close* method of the operator. The operator then, will communicate with the *LXReader thread*, so the latter can stop the waiting iterator, close the database connection by terminating the *KiviSession*, and finally release all open resources.

## 6.4 Implementation of Flink source abilities

In the previous subsection, we provided documentation on how our *source* connector works along with the Apache Flink streaming processing framework. We used a simple example that our data user needs to retrieve all information stored in a table. However, a most real scenario will be for the data user to request data concerning only specific fields of the dataset, or put some filter criteria to get a subset of the overall dataset. Such a scenario can be depicted in the following code snippet:

```
testFromSourceTable = tEnv.from("SENTIMENTS ").select($"id_str",
    $"sentiment_score").as("this_is_the_name");
testFromSourceTable =
testFromSourceTable.filter($"sentiment").isEqual("sentimentB").filter($"sentiment_score").isGreater(20.0);
TableResult tableResult = testFromSourceTable.execute();
```

Here, our data user needs to select the `id_str` and `sentiment_score` fields from the data table, whose `sentiment` value is equal to a given one, and whose score is greater than a given value. With our base implementation that was demonstrated in the previous subsection, Apache Flink would have needed to get the overall dataset from the INFINISTORE and apply the *project* and *filter* operations defined in this code snippet in-memory, inside Flink. This is very inefficient as the overall dataset would have needed to be transmitted from the data source and then Flink would have needed to calculate the result using its internal operations in-memory. To avoid this, the Flink *source* connector provides the *abilities* interfaces that allow custom implementations to accept operations from Flink and execute these operations in the data management layer. In our connector, we have implemented the following *abilities*, as depicted in Figure 6.

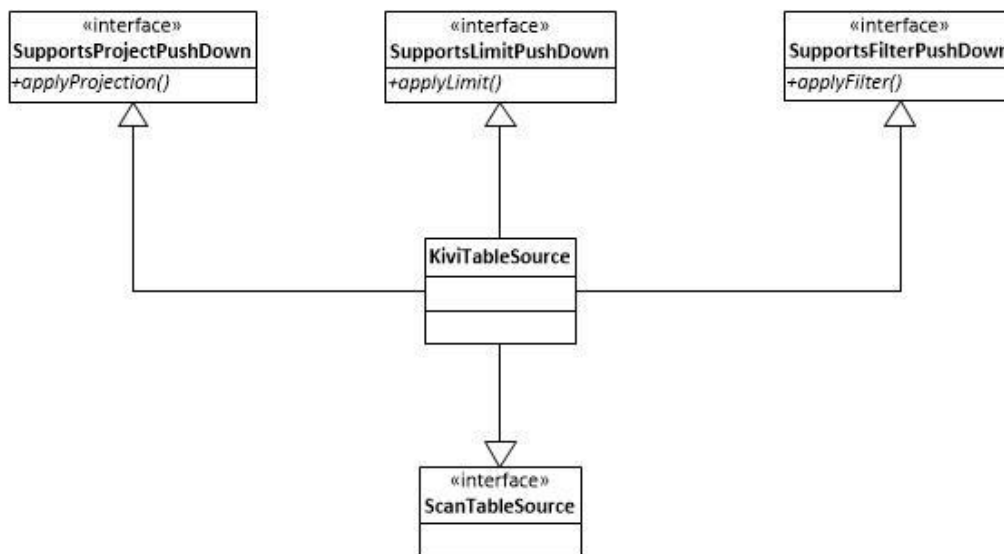


Figure 6: Flink Connector – The Source implementation: abilities

The *SupportsLimitPushDown* interface defines a method to apply this operator. It passes the number of the *limit* operation, so that the implementation can only return this given number of records. The source code of our *KiviTableSource* is illustrated in the following code snippet:

```
//methods that implements the SupportsProjectionPushDown interface
@Override
public boolean supportsNestedProjection() {
    return false;
}
@Override
```

```
public void applyProjection(int[][] ints) {
    this.projectionFields = ints;
}
```

The *SupportsProjectPushDown* interface defines a method to apply this operator. It passes the list of fields that the data user wants to receive, so that the implementation can only return these given columns. The source code of our *KiviTableSource* is illustrated in the following code snippet:

```
//methods that implement the SupportsLimitPushDown interface
@Override
public void applyLimit(long l) {
    this.limit = l;
}
```

The *SupportsFilterPushDown* interface defines the filter criteria that our *source* implementation needs to get into account in order to return only the data rows that satisfy these criteria. However, an external datastore could only support a subset of a filter operations provided by Flink. Due to this, it needs to examine the input filter criteria, inform the Flink Planner which can be supported and further pushed down to the storage, and which cannot and need to be handled by the Flink query engine itself. The source code of our *KiviTableSource* is illustrated in the following code snippet:

```
//methods that implement the SupportsFilterPushDown
@Override
public Result applyFilters(List<ResolvedExpression> list) {
    Result result = FilterTranslatorFactory.getSupported(list);
    this.acceptedFiltersResult = result;
    return result;
}
```

As has been illustrated, the input parameters passed by the implementations of the *abilities* interfaces are stored internally in the instance of our *KiviTableSource* to be further sent to the *KiviSourceFunction*, so that the latter can use them and create the query statement appropriately. For the *SupportsFilterPushDown*, we need however to further process the incoming input, that is sent in the form of an *ResolvedExpression*, decide which filter conditions are supported and transform these objects to what is meaningful for the external data source provider, our INFINISTORE. For that, we use a new factory that does this job for us, as depicted in the class diagram of Figure 7.

Here we follow the same approach and principles as in the implementation of our *Kafka Source Connector* described in D5.3 (“Library of Parallelized Incremental Analytics – III”). A *FilterTranslator* interface defines the method *translate* that all its implementations should provide. We have an implementation for each of the filter criteria the INFINISTORE’s storage engine supports: AND, OR, EQUALS (“=”), GREATER (“>”), GREATER\_OR\_EQUALS (“=>”), NOT NULL etc. Each of these classes transforms the input *ResolvedExpression* to a *Filter* that will be passed to the INFINISTORE via the *KiviSession* of the *LXReader*. The *FilterTranslatorFactory* receives the input list of expressions, validates which are supported, and finally, translates the supported ones to an object of the *Filter* class of the direct API of the storage engine of the INFINISTORE.

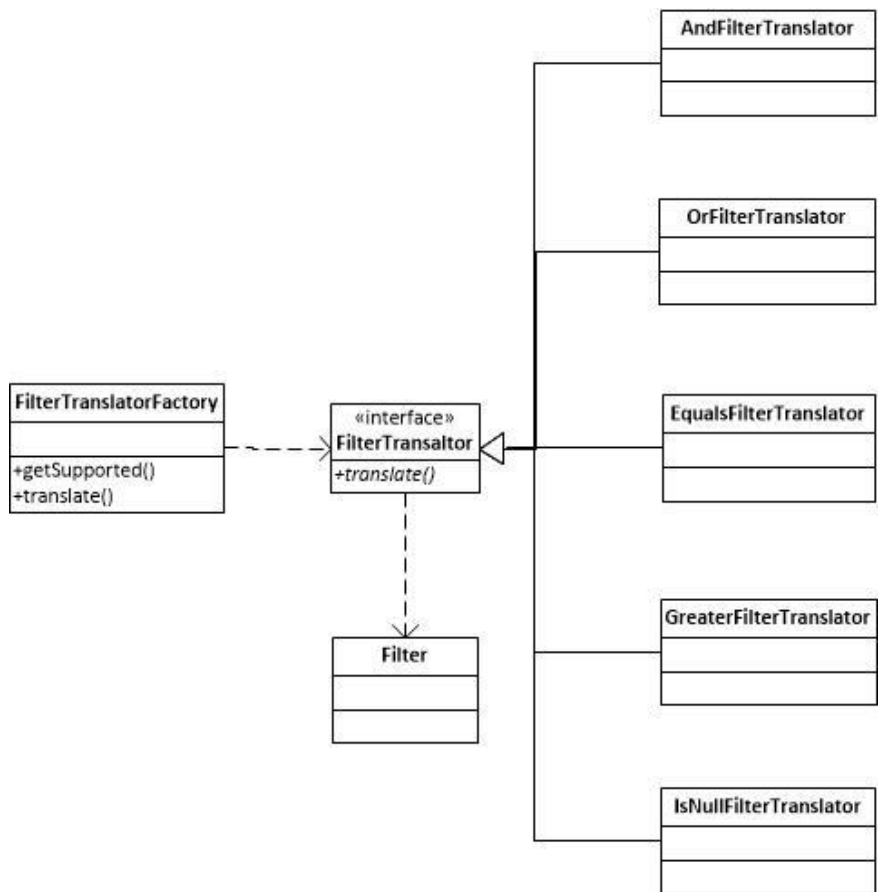


Figure 7: Flink Connector – The Source abilities: SupportsFilterPushDown

## 6.5 Demonstration of INFINSTORE Flink Connector

In this section we will provide some examples with client source code that the data user can make use of our provided Flink connector to access the INFINSTORE, make some queries and insert data using its *sink* part.

In the following code snippet, we have created a unit test that firstly opens a database connection with the INFINSTORE using its direct API, and adds 100 rows. Then, in the main method of this test, we create two materialized view: one for the source, and one for the sink. We will first read the everything from the source and the TableResult that will be obtained, will be written in the materialized view that is linked with the sink. Finally, we will read from the INFINSTORE and we will validate that these 100 rows are actually stored in the datastore. The code snippet is as follows:

```

package com.leanxcale.connector.flink;

import static java.util.Arrays.asList;

import com.leanxcale.connector.commons.test.LXBaseTest;
import com.leanxcale.exception.LeanxcaleException;
import com.leanxcale.kivi.database.Database;
import com.leanxcale.kivi.database.Field;
import com.leanxcale.kivi.database.Table;
import com.leanxcale.kivi.database.Type;
import com.leanxcale.kivi.session.Session;
import com.leanxcale.kivi.session.SessionFactory;
import com.leanxcale.kivi.tuple.Tuple;
    
```

```

import java.io.IOException;
import java.util.stream.IntStream;
import org.apache.flink.table.api.EnvironmentSettings;
import org.apache.flink.table.api.TableEnvironment;
import org.apache.flink.table.api.TableResult;
import org.junit.Assert;
import org.junit.BeforeClass;
import org.junit.Test;

public class HappyPathITest extends LXBaseTest{

    private static final String TABLE_SOURCE = "FLINK_TEST_SOURCE";
    private static final String TABLE_SINK = "FLINK_TEST_SINK";
    private static final int ROWS = 100;

    @BeforeClass
    public static void setup() throws IOException, LeanxcaleException, InterruptedException {
        try (Session session = SessionFactory.newSession(settings)) {
            Database database = session.database();

            database.createTable(TABLE_SOURCE, asList(new Field("FIELD1", Type.INT)),
                asList(new Field("FIELD2", Type.STRING), new Field("FIELD3", Type.DOUBLE)));

            database.createTable(TABLE_SINK, asList(new Field("FIELD1", Type.INT)),
                asList(new Field("FIELD2", Type.STRING), new Field("FIELD3", Type.DOUBLE)));

            Table table = database.getTable(TABLE_SOURCE);
            Tuple tuple = table.createTuple();
            IntStream.range(0, ROWS).forEach(x->{
                tuple.put("FIELD1", x);
                tuple.put("FIELD2", "field2 text:"+x);
                tuple.put("FIELD3", x+(x/10.0));
                table.insert(tuple);
            });

            session.commit();
            if (runningOnBlade) Thread.sleep(1000);
        }
    }

    @Test
    public void happyPathTest() throws IOException, LeanxcaleException, InterruptedException {
        EnvironmentSettings settings = EnvironmentSettings.inStreamingMode();
        TableEnvironment tEnv = TableEnvironment.create(settings);

        tEnv.executeSql("CREATE TABLE source (" +
            "    field1 INTEGER," +
            "    field2 VARCHAR," +
            "    field3 DOUBLE" +
            ") WITH (" +
            "    'connector' = 'kivi'," +
            "    'connection-url' = '"+ URL + "'," +
            "    'table-name' = '"+ TABLE_SOURCE + "'," +
            "    'bounded' = 'false'," +
            "    'stream-behaviour' = '"+ "end_on_null"+"' +
            ")");

        tEnv.executeSql("CREATE TABLE sink (" +
            "    field1 INTEGER," +
            "    field2 VARCHAR," +
            "    field3 DOUBLE" +
            ") WITH (" +
            "    'connector' = 'kivi'," +
            "    'connection-url' = '"+ URL + "'," +

```

```

        "    'table-name' = '"+ TABLE_SINK +"'," +
        "    'commit-batch' = '"+ 30 +"'" +
        "  )");

    org.apache.flink.table.api.Table source = tEnv.from("source");
    org.apache.flink.table.api.Table table = tEnv.sqlQuery("select * from source");
    TableResult result = table.executeInsert("sink");

    TableResult tableResult = testTable.execute();

    tableResult.print();

    tableResult.collect().forEachRemaining((t) -> {
        System.out.println(t);
    }));

    List<Row> rows = new ArrayList<Row>();
    result.collect().forEachRemaining(rows::add);
    Assert.assertEquals(ROWS, rows.size());

    long sinkCount = getTableRowCount(TABLE_SINK);
    Assert.assertEquals(ROWS, sinkCount);
}

private long getTableRowCount(String tableName) throws IOException, LeanxcaleException {
    try (Session session = SessionFactory.newSession(settings)) {
        Table table = session.database().getTable(tableName);
        return table.find().asStream().stream().count();
    }
}
}
}

```

In the next code snippet, we will validate our implementation for the support of the push down of the projections. Again, we will add 100 rows in a table in INFNISTORE and then we will read from the datastore, applying the *select* criteria that will be translated into *projections*. Finally, we will read using our *source connector* and the unit test will validate that we can read only the specific columns requested. If we try to read something not included in the projection list, then an error should be thrown. This code snippet is as follows:

```

package com.leanxcale.connector.flink;

import static java.util.Arrays.asList;

import com.leanxcale.connector.commons.test.LXBaseTest;
import com.leanxcale.exception.LeanxcaleException;
import com.leanxcale.kivi.database.Database;
import com.leanxcale.kivi.database.Field;
import com.leanxcale.kivi.database.Table;
import com.leanxcale.kivi.database.Type;
import com.leanxcale.kivi.session.Session;
import com.leanxcale.kivi.session.SessionFactory;
import com.leanxcale.kivi.tuple.Tuple;
import java.io.IOException;
import java.util.stream.IntStream;
import org.apache.flink.table.api.EnvironmentSettings;
import static org.apache.flink.table.api.Expressions.$;
import org.apache.flink.table.api.TableEnvironment;
import org.apache.flink.table.api.TableResult;
import org.junit.Assert;
import org.junit.BeforeClass;
import org.junit.Test;

/**

```



```

*
* @author Pavlos Kranas (LeanXcale)
*/
public class SupportsProjectionPushDownTest extends LXBaseTest{

    private static final String TABLE_NAME_OFFSET = "_SUPPORTS_PROJECT_PUSHDOWN";
    private static final String TABLE_SOURCE = "FLINK_TEST_SOURCE" + TABLE_NAME_OFFSET;
    private static final int ROWS = 100;

    @BeforeClass
    public static void setup() throws IOException, LeanxcaleException, InterruptedException {
        try (Session session = SessionFactory.newSession(settings)) {

            Database database = session.database();

            database.createTable(TABLE_SOURCE, asList(new Field("FIELD1", Type.INT),
                asList(new Field("FIELD2", Type.STRING), new Field("FIELD3", Type.DOUBLE)));

            Table table = database.getTable(TABLE_SOURCE);
            Tuple tuple = table.createTuple();
            IntStream.range(0, ROWS).forEach(x->{
                tuple.put("FIELD1", x);
                tuple.put("FIELD2", "field2 text:"+x);
                tuple.put("FIELD3", x+(x/10.0));
                table.insert(tuple);
            });

            session.commit();
            if (runningOnBlade) Thread.sleep(1000);
        }
    }

    @Test
    public void happyPathSQLTest() throws IOException, LeanxcaleException, InterruptedException {
        EnvironmentSettings settings = EnvironmentSettings.inStreamingMode();
        TableEnvironment tEnv = TableEnvironment.create(settings);

        tEnv.executeSql("CREATE TABLE source (" +
            "    field1 INTEGER," +
            "    field2 VARCHAR," +
            "    field3 DOUBLE" +
            ") WITH (" +
            "    'connector' = 'kivi'," +
            "    'connection-url' = '"+ URL + "'," +
            "    'table-name' = '"+ TABLE_SOURCE +"'" +
            ")");

        org.apache.flink.table.api.Table table = tEnv.sqlQuery("select field1, field2 from source");
        TableResult tableResult = table.execute();
        tableResult.collect().forEachRemaining((t) -> {
            Assert.assertEquals(2, t.getArity());
            Assert.assertNotNull(t.getField("field1"));
            Assert.assertNotNull(t.getField("field2"));
            try {
                t.getField("field3");
            } catch (Exception ex) { /*ok*/ }
        });
    }

    @Test
    public void happyPathNativeTest() throws IOException, LeanxcaleException, InterruptedException {

```

```

EnvironmentSettings settings = EnvironmentSettings.inStreamingMode();
TableEnvironment tEnv = TableEnvironment.create(settings);

tEnv.executeSql("CREATE TABLE source (" +
    "    field1 INTEGER," +
    "    field2 VARCHAR," +
    "    field3 DOUBLE" +
    ") WITH (" +
    "    'connector' = 'kivi'," +
    "    'connection-url' = '"+ URL + "'," +
    "    'table-name' = '"+ TABLE_SOURCE + "'" +
    ")");

org.apache.flink.table.api.Table table = tEnv.from("source").select($"field1"),
$"field3").as("other_name");
TableResult tableResult = table.execute();
tableResult.collect().forEachRemaining((t) -> {
    Assert.assertEquals(2, t.getArity());
    Assert.assertNotNull(t.getField("field1"));
    Assert.assertNotNull(t.getField("other_name"));
    try {
        t.getField("field2");
    } catch (Exception ex) { /*ok*/ }
});
}
}

```

The last code snippet will make a combination of both *projections* and *limit* operations, to firstly validate that the support for *limit* works and the Flink query planner works when creating its internal query execution plan, examining various scenarios. That will involve the invocation of the *copy* method of our *KiviTableSource* that needs to preserve the newly added instructions for pushing down combination of operators. The structure of the test is similar to the previous ones and the source code of our unit test is illustrated in the following code snippet:

```

package com.leanxcale.connector.flink;

import com.leanxcale.connector.commons.test.LXBaseTest;
import com.leanxcale.exception.LeanxcaleException;
import com.leanxcale.kivi.database.Database;
import com.leanxcale.kivi.database.Field;
import com.leanxcale.kivi.database.Table;
import com.leanxcale.kivi.database.Type;
import com.leanxcale.kivi.session.Session;
import com.leanxcale.kivi.session.SessionFactory;
import com.leanxcale.kivi.tuple.Tuple;
import java.io.IOException;
import java.util.ArrayList;
import static java.util.Arrays.asList;
import java.util.List;
import java.util.stream.IntStream;
import org.apache.flink.table.api.EnvironmentSettings;
import static org.apache.flink.table.api.Expressions.$;
import org.apache.flink.table.api.TableEnvironment;
import org.apache.flink.table.api.TableResult;
import org.apache.flink.types.Row;
import org.junit.Assert;
import org.junit.BeforeClass;
import org.junit.Test;

/**
 *
 * @author Pavlos Kranas (LeanXcale)
 */

```

## D3.8 – Data Streaming and Data at Rest Queries Integration - III

```
public class SupportsLimitPushDownTest extends LXBaseTest{

    private static final String TABLE_NAME_OFFSET = "_SUPPORTS_LIMIT_PUSHDOWN";
    private static final String TABLE_SOURCE = "FLINK_TEST_SOURCE" + TABLE_NAME_OFFSET;
    private static final int ROWS = 100;

    @BeforeClass
    public static void setup() throws IOException, LeanxcaleException, InterruptedException {
        try (Session session = SessionFactory.newSession(settings)) {

            Database database = session.database();

            database.createTable(TABLE_SOURCE, asList(new Field("FIELD1", Type.INT)),
                asList(new Field("FIELD2", Type.STRING), new Field("FIELD3", Type.DOUBLE)));

            Table table = database.getTable(TABLE_SOURCE);
            Tuple tuple = table.createTuple();
            IntStream.range(0, ROWS).forEach(x->{
                tuple.put("FIELD1", x);
                tuple.put("FIELD2", "field2 text:"+x);
                tuple.put("FIELD3", x+(x/10.0));
                table.insert(tuple);
            });

            session.commit();
            if (runningOnBlade) Thread.sleep(1000);
        }
    }

    @Test
    public void pushDownLimitTest() throws IOException, LeanxcaleException, InterruptedException {
        EnvironmentSettings settings = EnvironmentSettings.inStreamingMode();
        TableEnvironment tEnv = TableEnvironment.create(settings);

        tEnv.executeSql("CREATE TABLE source (" +
            "    field1 INTEGER," +
            "    field2 VARCHAR," +
            "    field3 DOUBLE" +
            ") WITH (" +
            "    'connector' = 'kivi'," +
            "    'connection-url' = '"+ URL + "'," +
            "    'table-name' = '"+ TABLE_SOURCE +"'" +
            ")");

        org.apache.flink.table.api.Table table = tEnv.from("source").limit(5);
        TableResult tableResult = table.execute();
        List<Row> list = new ArrayList<>(100);
        Thread.sleep(5000); //wait for source to poll the results first
        tableResult.collect().forEachRemaining(t -> {
            list.add(t);
        });

        Assert.assertEquals(5, list.size());

        table = tEnv.from("source").limit(8, 15);
        tableResult = table.execute();
        List<Row> listOther = new ArrayList<>(100);

        Thread.sleep(5000); //wait for source to poll the results first
        tableResult.collect().forEachRemaining(t -> {
            listOther.add(t);
        });
    }
}
```

```

    Assert.assertEquals(15, listOther.size());
}

@Test
public void pushDownLimitWithProjectionTest() throws IOException, LeanxcaleException,
InterruptedException {
    EnvironmentSettings settings = EnvironmentSettings.inStreamingMode();
    TableEnvironment tEnv = TableEnvironment.create(settings);

    tEnv.executeSql("CREATE TABLE source (" +
        "    field1 INTEGER," +
        "    field2 VARCHAR," +
        "    field3 DOUBLE" +
        ") WITH (" +
        "    'connector' = 'kivi'," +
        "    'connection-url' = '"+ URL + "'," +
        "    'table-name' = '"+ TABLE_SOURCE + "'" +
        ")");

    org.apache.flink.table.api.Table table = tEnv.from("source").select($"field1",
    $"field2").limit(5);
    TableResult tableResult = table.execute();
    List<Row> list = new ArrayList<>(100);

    Thread.sleep(5000); //wait for source to poll the results first
    tableResult.collect().forEachRemaining(t -> {
        list.add(t);
    });

    Assert.assertEquals(5, list.size());
    for(Row row : list) {
        row.getField("field1");
        row.getField("field1");
        try {
            row.getField("field3");
        } catch(Exception ex) { /*ok*/}
    }
}
}

```

## 7 Combined Data Streaming and Data at Rest Illustration

Having discussed how the INFINITECH solution handles efficient combination of data in-flight and data at rest in the previous section, it is valuable to provide an illustration of how this would add value with respect to a specific INFINITECH Pilot. Hence, in this section we provide a walk-through of an updated Flink topology for the real-time risk assessment in investment banking (Pilot#2) use-case (also discussed previously in Section 2.2). Recall that the aim of this use-case is to produce as close to real-time risk assessments for customer investment portfolios as possible, e.g. such that customers can be alerted if significantly elevated risks start to be observed. Risk is quantified with the metric Value-at-Risk (VaR), which is calculated by analysing the asset returns over many time periods and calculating an average % loss across the worst 5% performing time periods (as a worst-case scenario).

To understand the issues with calculating real-time VaR, we first provide an illustration of a naive solution in Figure 2. In this setup, we take as input on the left-hand side a stream of asset pricing data (e.g. from a stock market or trading platform). To calculate VaR for an asset at the current moment, we need to retrieve the historical pricing data for that asset, which will involve an expensive table scan over whatever back-end database is being used to persistently store asset prices. Once this is done, the resultant array of asset prices will need to be transferred over the network from the database with the asset pricing data to the current Flink Worker responsible for processing the new asset price event (that again may be expensive). The historical prices will then be merged into the stream with the new price event and sent on to an aggregator to calculate asset returns for each time window being considered. If we are unlucky here, this may involve another expensive network transfer, depending on if the transformer and window aggregator are co-located on the same machine. Once the asset returns for each window are calculated, the results are sent onward to VaR calculation, which produces a single score for the asset based on the updated data. It is worth noting at this stage that all of this work will have been for naught if the initiating asset price point does not contribute to a window in the worst 5% (as the VaR calculation only cares about that 5%). Finally, assuming that the VaR value for the asset has changed, then the new value will be sent to another transformer, to calculate aggregate VaR for each customer’s portfolio that holds the asset.

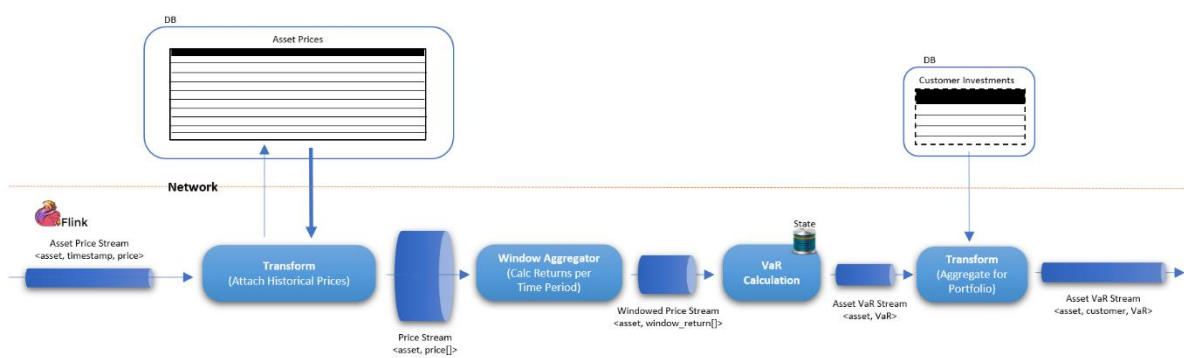


Figure 8: Naive VaR Flink Topology

To summarize, there are four principles that should be followed when designing a good solution to this problem, which the above design fails under:

1. We want to minimise the amount of processing that occurs when each asset price update arrives, since this happens very frequently.

2. We should only trigger the re-calculation of VaR if we have good reason to believe that this will result in the VaR score for an asset to change. Practically, this means we should only trigger calculation at the end of a time window, and only if the current window is part of the worst 5%.
3. We should avoid expensive table scans over large tables.
4. We should avoid transferring large amounts of data over the network.

Given these principles, let us now consider a superior topology that utilises the INFINITECH streaming engine to solve these issues. Figure 3 illustrates an alternative Flink topology solution that integrates dynamic tables and continuous queries over the INFINITECH data layer (i.e., the INFINISTORE). As with the previous topology, we ingest a stream of asset pricing updates over time as a stream. However, instead of using this as a trigger to start the process of recalculating VaR, we instead assign time window labels that we can use to structure processing up-front. With these labels in place, we can then leverage the new remote dynamic table functionality provided by the INFINISTORE to ‘push-down’ both the storage of the asset pricing data and the incremental calculation of asset returns at the end of each time window into the store itself. In this way, we never need to transfer a large batch of asset pricing data across the network. Moreover, the INFINISTORE in conjunction with Flink’s continuous query semantics means that processing is incremental and localized to only the window of current interest, avoiding large table scans and redundant computation. Finally, dynamic tables can also be easily pipelined back into stream processing within Flink, where the ‘events’ emitted are the updates made to the table. In this case, an update to the Dynamic Returns Per Time Window Table indicates that the return for an asset has been calculated for a new time window, which acts as a more useful trigger for re-calculation of VaR for that asset. Overall, this is a much more efficient and scalable topology that can handle high volume streams of asset prices, as well as parallelism in computation across assets.

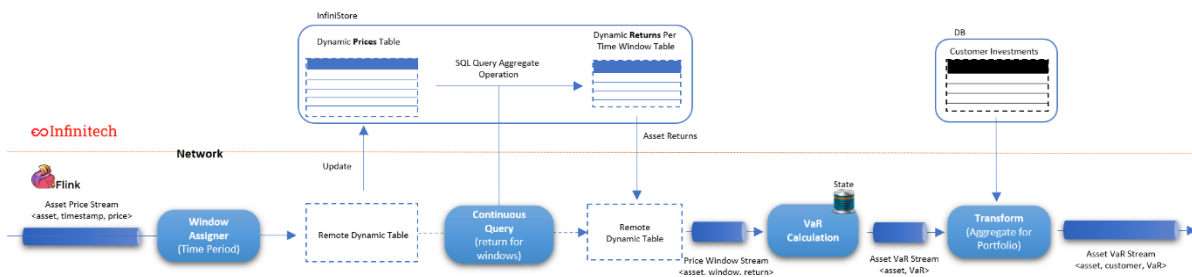


Figure 9: VaR Calculation Topology using the Infnitech Streaming Engine

## 8 Conclusions

This report documented the work that has been carried out in the scope of task T3.3 “Integrated Querying of Streaming Data and Data at Rest” at this phase of the project. The main objective of this task is to implement a data framework that can provide a unified manner for accessing data that can be considered both *streaming* and *data-at-rest* at the same time, thus allowing the correlation of stream and batch processing in an effective way. This is very important as it will remove the current obstacles and limitations of existing solutions that promise to deliver this functionality, however they fail due to the inherent barriers for accessing static data effectively so that they can be used in real-time for streaming processing. As we explained, the high rate of data ingestion that comes from a stream cannot be served by traditional data management systems, and as a result, most of the system integrators are using data queues as an interim layer for pushing data from a stream, while a consumer process periodically takes data from the queue and ingest them to the datastore in what is called *micro batches*. This has the drawback that data analysts cannot apply their AI algorithms in real-time and in fact, they can only provide *near* real-time business intelligence. Another drawback is the need to combine streaming data with information that has been retrieved from aggregation operations over the static data, which are time consuming. Modern solutions usually *cache* this information so that it can be instantly available to the stream processing engine, with the drawback of losing data consistency, when the dataset is being frequently modified at the same time.

This report firstly provided an analysis over the state-of-the-art of streaming processing frameworks, and analyzed the current status of this ecosystem. We decided to use Apache Flink as the core of the INFINITECH unified query processing framework, as it is a popular solution with lots of documentation that also provides some very important functionalities: it has been initially designed without the support of stateful operations and fault-tolerance, so that it can be easily scaled out. At its current version, it provides the support of additional operators that are stateful, and defines several levels of APIs for context event processing on the very low level, for streaming processing using *DataStreams* and the ability to write SQL statements for query processing on its higher level *Table* API. Tables can be transformed to data streams and vice versa, thus allowing for the correlation of streaming and data processing. However, the implementation of the stateful operations requires that objects reside in memory and being updated by the Apache Flink framework, with all the aforementioned barriers.

As the scope of this task is to provide the integrated query processing framework of the INFINITECH platform, it became obvious that our framework will have to rely on the unique characteristics of the data management layer of the platform itself: the support of hybrid transactional and analytical processing, the online aggregations and the polyglot capabilities. The highly scalable transactional management of the data repository allows for data ingestion in very high rates, which is what a streaming channel requires. We can remove now the interim data queue and insert data directly to the storage. But this is not the only benefit from using our own data storage layer. HTAP allows for performing analytical query processing on live data, as they are being modified by operational workloads, and as a result, gives the ability for executing AI algorithms on real data. Real time BI can be now achieved by delegating the need for maintaining Flink’s *materialized views* down to the data table of the datastore. This ensures data consistency, as the transactional semantics are provided by the database itself, rather than relying on the streaming framework which simply maintains the sequence of the modified operations, but does not guarantee the serializability of the order of execution (in terms of a database transaction). Moreover, it allows for the streaming engine to effectively scale out, which cannot be done when sharing content such as *materialized views* between different Flink *sessions*. Additionally, the online aggregations of INFINITECH allow to execute aggregation statements (i.e., average, summary, count, etc.) with a complexity of  $O(1)$  instead of  $O(n)$ , as supposed by the need for scanning the entire dataset. Therefore, the streaming operators can directly query the datastore for such information, instead of caching those values and lose the consistency of the data.

In order to integrate the components of INFINITECH that provide those characteristics with Apache Flink, we designed and implemented the INFINITECH Flink *connector* that implements those operations that can be used by the framework. We achieved to move the state of the stateful Flink operations down to the data

management layer, so that those operations can be now considered stateless and can easily scale out. Flink now does not have to maintain any state and can be scale out independently, while the data management layer has been also designed to scale horizontally, as explained in the deliverables of task T3.1. Therefore, we claim that we have no data bottleneck in our solution. Finally, as the definition of the pilot needs and requirements for their integrated solutions have matured during this phase of the project, we took the generic use case described in the first version of this document, and made it more concrete, trying to address the needs for a specific pilot, pilot#2, of the INFINITECH project. We illustrated how our solution can remove 4 important technological obstacles that this pilot would have normally faced.

Finally, it is important to highlight the interconnection between different tasks of the technical WPs of INFINITECH. The technology implemented in this task makes use of the high rate data ingestion capabilities implemented in T3.1, the polyglot query processing implemented in T3.2, the online aggregates implemented in T5.3 and the incremental analytics implemented in T5.2. Moreover, it is the base for the technological advancements implemented in T3.4. Technological outcomes of this task, like the Flink Operator, have been already incorporated into LeanXcale’s product, opening new opportunities to establish Proof-of-Concepts with new potential customers in the field of streaming query processing.

Table 3: Conclusions (TASK Objectives with Deliverable achievements)

Objectives	Comment
<i>produce a unified framework for querying streaming data and data at rest</i>	The proposed solution offers a framework that allows integrated query processing over streaming data and data <i>at-rest</i> , using Apache Flink as the core, and our INFINISTORE Flink connector as the bridge to the data management layer.
<i>provide an SQL-based framework to correlate streaming tuples with the contents of the database</i>	With the use of the Table API, the user can define materialized views and submit SQL relational algebraic operations, that will be pushed down to the database, via our Flink connector.
<i>make use of correlated streaming and static data to update the database based on the contents of the incoming streams.</i>	In this task, we went a step beyond this objective, as we are not only capable of correlated streaming and static data to update the database content, but the database itself can send a stream of data, using the incremental analytics developed under T5.2 to further allow the streaming processing framework to make use of this in its deployed operators
<i>provide the means for querying streaming data and data at rest in an integrated fashion</i>	We relied on a popular streaming processing framework at the core of the INFINITECH offering in order to have this integrated query processing and we’ve prepared recipes for the deployment using the INFITECH way of deployments (defined in WP6) to allow the ease integration with other INFINITECH offerings



Table 4: Conclusions – (map TASK KPI with Deliverable achievements)

KPI	Comment
<i>Increase in parallelization of stateful analytics</i>	<p><i>Target Value</i> &gt; = 100%</p> <p>Even if this KPI is not targeting the work carried out under this task, its outcomes however will be used as the technology pillar of the work that is currently carried out under the scope of T3.4 that this KPI is targeting. Towards this, our integrated query processing framework based on the Apache Flink integrated with the INFINISTORE and its already adapted innovations, allow the data user to push down to the database all data intensive operators, removing the latency, being able to scale out effectively, and removing the state from the deployed operators. This allows to build stateful analytics, that can be parallelized, as their state can be now handled by the datastore itself.</p>

## 9 References

- [1] Arasu A. et al. (2016) STREAM: The Stanford Data Stream Management System. In: Garofalakis M., Gehrke J., Rastogi R. (eds) Data Stream Management. Data-Centric Systems and Applications. Springer, Berlin, Heidelberg
- [2] Çetintemel U. et al. (2016) The Aurora and Borealis Stream Processing Engines. In: Garofalakis M., Gehrke J., Rastogi R. (eds) Data Stream Management. Data-Centric Systems and Applications. Springer, Berlin,
- [3] F. Khan, N. Akhtar and M. A. Qadeer, "RFID Enhancement in Road Traffic Analysis by Augmenting Receiver with TelegraphCQ," 2009 Second International Workshop on Knowledge Discovery and Data Mining, Moscow, 2009, pp. 331-334.
- [4] Park, Hoyong, Eric Hsiao, and Andy Piper. "Continuous query language (CQL) debugger in complex event processing (CEP)." U.S. Patent No. 9,329,975. 3 May 2016.
- [5] D. J. Abadi et al., "The Design of the Borealis Stream Processing Engine," p. 13.
- [6] Cranor C.D., Johnson T., Spatscheck O. (2016) Stream Processing Techniques for Network Management. In: Garofalakis M., Gehrke J., Rastogi R. (eds) Data Stream Management. Data-Centric Systems and Applications. Springer, Berlin, Heidelberg C.
- [7] M. Hirzel et al., "IBM Streams Processing Language: Analyzing Big Data in motion," in IBM Journal of Research and Development, vol. 57, no. 3/4, pp. 7:1-7:11, May-July 2013.
- [8] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente and P. Valdúriez, "StreamCloud: An Elastic and Scalable Data Streaming System," in IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 12, pp. 2351-2365, Dec. 2012.
- [9] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed Stream Computing Platform," in 2010 IEEE International Conference on Data Mining Workshops, 2010, pp. 170–177.
- [10] "Apache Storm.", <http://storm.apache.org/>.
- [11] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K. L. Wu, "Elastic scaling of data parallel operators in stream processing," in 2009 IEEE International Symposium on Parallel Distributed Processing, 2009, pp. 1–12.
- [12] "Spark Streaming | Apache Spark.", <https://spark.apache.org/streaming/>.
- [13] "Stratosphere » Next Generation Big Data Analytics Platform." <http://stratosphere.eu/>
- [14] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink™: Stream and Batch Processing in a Single Engine," p. 12.
- [15] "Samza.", <http://samza.apache.org/>.
- [16] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A Timely Dataflow System," in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, New York, NY, USA, 2013, pp. 439–455.
- [17] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Making State Explicit for Imperative Big Data Processing," presented at the USENIX ATC'14: 2014 USENIX Annual Technical Conference, Philadelphia, USA, 2014.
- [18] "Amazon Kinesis.", <https://aws.amazon.com/kinesis/>.
- [19] "WSO2 CEP.", <https://wso2.com/products/complex-event-processor/>.
- [20] "FlinkCEP.", <https://ci.apache.org/projects/flink/flink-docs-release-1.11/dev/libs/cep.html>.
- [21] "Apache Calcite", <https://calcite.apache.org/>