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

Infinitech

D3.4 – Integrated (Polyglot) Persistence- I

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

HORIZON 2020 - ICT-11-2018

This project has received funding from the European Union's horizon 2020 research and innovation programme under grant agreement no 856632

Contributing Partners

Partner Acronym	Role[1]	Name Surname[2]
UBI	Lead Beneficiary	Konstantinos Perakis
LXS	Contributor	Ricardo Jiménez-Peris, Boyan Kolev, Javier López Moratalla, Patricio Martinez, Sandra Ebro, Francisco Ballesteros, Rogelio Rodriguez
UPRC	Contributor	Ioannis Kranas
UBI	Contributor	Konstantinos Perakis, Dimitris Miltiadou
GRAD	Internal Reviewer	Marta Sestelo, Borja Pintos
GFT	Internal Reviewer	Ernesto Troiano
INNOV	Quality Assurance	Dimitris Drakoulis

Revision History

Version	Date	Partner(s)	Description
0.1	2020-07-28	UBI, LXS	ToC Version
0.2	2020-07-28	UBI, LXS	Input on Executive summary, Introduction
0.3	2020-07-28	LXS, UPRC	Input on Section 2
0.4	2020-07-29	LXS, UBI	Input on Section 3
0.5	2020-07-29	LXS, UBI	Input on Section 4
0.6	2020-07-30	LXS, UPRC	Input on Section 5
0.7	2020-07-31	UBI, LXS, UPRC	Input on Conclusions
1.0	2020-07-31	UBI. LXS	First Version for Internal Review
1.1	2020-08-05	GRAD	Internal Review
1.2	2020-08-24	GFT	Internal Review
1.3	2020-08-24	UBI	Addressing review comments
1.4	2020-08-25	UBI, LXS	Finalizing the document
2.0	2020-08-25	UBI	Version for Quality Assurance
3.0	2020-08-28	UBI	Version for Submission

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

[2] Can be left void

Executive Summary

The goal of task T3.2 “Polyglot Persistence over BigData, IoT and Open Data Sources” is to provide a common and integrated way to access data that is stored in a structured, semi-structured or even unstructured fashion over a variety of heterogeneous data stores, in a unified manner. The rationale behind this task is that in modern enterprises, especially in organizations coming from the finance and insurance sectors, it is a necessity to access and process data coming from multiple sources. As there is no *one size fits all* data management system, different databases have been proposed to serve different types of needs and workloads. Hence, task T3.2 aims to provide a solution for efficient and unified integrated access over a variety of heterogeneous data stores that provide data in all structured, unstructured or semi-structured fashion. As an integral part of the INFINITECH data management layer, its goal is to provide a common API that can be used to access data seamlessly, hiding the internal complexities from the data scientists and application developers, therefore providing *polyglot* capabilities to the platform.

Towards this end, a state-of-the-art analysis of the existing frameworks that can be considered as *polystores* has been conducted first, aiming to identify the best practices that have been proposed in the literature and are widely used by modern enterprises today, and to highlight their weakness and current challenges that are open to research to further improve their solutions. Moreover, as different datastores often provide their own query language for accessing and retrieving data one of the requirements for the polyglot data management system that we are presenting in this deliverable is to provide a seamless way to access data coming from a variety of heterogeneous sources. In this context, we have defined the INFINITECH Common Query Language that will be used by the data scientists to request data for analysis in a unified manner. The INFINITECH Common Query Language has a lot of similarities with the standard SQL, allowing the data scientists to write queries in well-known standards. However, it also allows writing in the native language of a data store, in cases the user wants to exploit some unique characteristics of the database that cannot be expressed with standard SQL.

In addition to the INFINITECH Common Query Language definition, we have concluded on the principle architectural design pattern that will drive the whole design of the solution, namely the INFINITECH Integrated Polystore Engine, and the main building blocks and components of the solution have been also designed. We follow the mediator-wrapper approach, where one orchestrator drives the whole execution of the query, and different wrappers implement the data connectivity and query execution on the target data stores, hiding the internal complexity and becoming transparent from the orchestrator. The orchestrator will be part of the central data repository and data management layer of INFINITECH, and will extend it with the aim to provide the polyglot capabilities. It will be incorporated with the query engine of the central data repository of the platform, with the aim of taking advantage of the already available functionalities that are crucial for the effective data retrieval: the query planner, query optimizer and query executor. All of them will be further extended with a view to take into account the polyglot capabilities that the engine will now offer.

Finally, it is important to highlight that this is the first version of this deliverable that reports the work that has been done in the scope of task T3.2 until M11. At this phase of the project, apart from the necessary initial state-of-the-art analysis of other polystore systems, most of the effort was spent in the definition of the INFINITECH Common Query Language, which is the basis for this task. A great effort was also spent to design the basic architecture of the INFINITECH Integrated Polystore Engine solution and the fundamentals that will drive the implementation and the more advanced functionalities that have been planned for the next periods. At this phase, we also provide a basic implementation of the solution, allowing for a query execution over federated databases: the central

data repository of INFINITECH and a relational JDBC-compatible database. This was done as a means to validate our design and to create a roadmap on how to incorporate other NoSQL databases and Hadoop[3] data lakes. As there will be two more versions of this deliverable, a more complete solution is planned to be presented in the second version, while in the third, we plan to present the overall solution, along with demonstrators and results.

[3] https://hadoop.apache.org/

Table of Contents

List of Figures

Abbreviations

API	Application Programming Interface
BDVA	Big Data Value Association
DoA	Description of Action
HDFS	Hadoop Distributed File System
HTAP	Hybrid Transactional and Analytical Processing
IoT	Internet of Things
JDBC	Java DataBase Connectivity
JSON	Javascript Object Notation
MFR	Map, Filter, Reduce operations
NoSQL	Not Structured Query Language
ODBC	Open DataBase Connectivity
RDBMS	Relational DataBase Management System
SQL	Structured Query Language
WP	Work Package

1. Introduction

This deliverable summarizes the work that has been done in the scope of task T3.2 Polyglot Persistence over BigData, IoT and Open Data Sources” during the first phase of the project (M11). The goal of this task is to provide common and integrated access over structured, unstructured or semi-structured data that can be stored in a variety of different and heterogeneous data management systems.

Nowadays, modern enterprises, including the ones originating from the finance and insurance sectors, are utilising, accessing and processing data from a various of heterogeneous sources which are stored in an emerging number of different data management systems designed to serve different type of requirements and workloads. In this sense, a finance organization may use, on the one hand, traditional relational datastores that ensure transactional semantics and provide data consistency, which store data in a structured schema: the relational model. On the other hand, there is always the need for using NoSQL databases that can scale more adequately, sacrificing the support for transactional semantics, and often using lesser strict data models, so that the data can be considered as semi-structured. Document-based datastores are a prominent example, where the schema of the data can be easily modified and extended so that it will be adapted in rapidly changed software. Key-value stores are also often used to store information coming from IoT devices, a pattern that is a norm in the insurance companies where sensor data from smart devices or agricultural sensors are being continuously ingested, or logging information tracking the web traffic or the finance transactions of a customer, which is of great interest for the finance sector. Key-value stores are candidates where there is a huge write-intensive workload with rare data modifications, where the data management system needs to scale out easily. Even if they do not provide any guarantees over the schema of the *value,* they are considered as semi-structured, as the data tends to preserve the schema. Finally, such organizations need to process data coming from external sources, like feeds from social media networks, or newspaper articles, towards the extraction of trends and opportunities. This type of data can be considered as unstructured and are often being imported into a data lake and use specific tools for data analysis that can extract this type of information.

Accessing and processing data coming from such a variety of heterogeneous data sources is a very complicated task at various levels. Firstly, each of the data stores provides specific methods for data connectivity, while at the same time, makes use of different query languages. Even different relational databases might provide different connectivity (i.e. JDBC vs ODBC) and even if they all rely on the SQL query language, their dialect might differ significantly. The data scientist must be aware and familiar with a variety of different tools and languages. Moreover, not all of the data base management systems return data in the same model. Relational databases return data as table with relationships, which can be easily transformed into an entity-relational model. However, key-value and NoSQL stores do not comply with specific schemas and the application developer or the data scientist must maintain different models and create a common layer on top. Most importantly, when there is the need to combine data coming from different sources, where joining data sources is mandatory, this must be done in the application or data processing level. However, joining data sources is a very challenging task and there is a lot of literature which has been implemented by the database management systems, on how to do it effectively. In fact, this operation should be transparent to the data scientist and application developer and should be implemented in the data management level that can do it more effectively. Due to this, various data analytical frameworks have been used during the recent years. Those frameworks provide a common way to address these challenges, however, they still require to fetch the majority of data into the data processing layer, which can become very resource consuming.

Towards this end, Task T3.2 will provide a unique interface which will be based on the JDBC specifications to make use of a native API and a common query language that is very close to SQL, as well as to make use of a native scripting language for query processing. By providing both a common API and scripting language for transparent data retrieval from heterogeneous data stores, INFINITECH will extend its data management system with polyglot capabilities, thus providing an integrated data layer that can be used transparently by the Open APIs and the semantic framework.

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 T3.2 at this phase of the project (M11). This task lasts until M27, and therefore, two more versions will be released, extending and modifying, when necessary, the content of this document, following the agile approach for system development and with the aim of updating the solution and implementation with the current trends of the environment as the project progresses. The work that has been done during this phase (M03-M11) was mainly focused on the definition of the INFINITECH Common Query Language, which is the basis of the whole solution. Moreover, the basic architectural design of the INFINITECH Integrated Polystore Engine solution and first proof-of-concept implementation has been provided towards the validation of our approach. According to the plan, in the second version of this document, more details will be reported on the design and implementation of the core of the solution, along with demonstrators regarding the installation and use of the solution, targeting a variety of other datastores that will have been integrated and supported at that phase. In the third and final version, it is planned that this component will have been fully integrated with the query engine of the central data repository of INFINITECH, thus being an integral part of the data management layer of the platform, and exploiting the already available functionalities for efficient query processing.

1.2. Insights from other Tasks and Deliverables

The work that is reported in this deliverable is based on the overview description of the corresponding task T3.2, which has been further specified in more details at the WP2 level, which is a fundamental work package that defines the overall requirements of the whole platform. More precisely, task T2.1 with the corresponding D2.1 deliverable refers to the user stories of the pilots that will be accommodated by the platform, and reports their user requirements. To this direction, task T2.3 with the corresponding D2.5 deliverable defines the specification of the technologies that INFINITECH provides, and translates the user stories and requirements to specific technical requirements that must be addressed by the technologies. Moreover, task T2.5 and deliverable D2.9 provide a list of data assets specifications, where the available target data store that needs to be taken into account is defined. Last but not least, the work that is reported in this document is also related with the Reference Architecture of the INFINITECH, as the solution provided by the Integrated Polystore Engine is an integral part of the platform. It is worth to mention that even if there is no direct connection with task T3.2 and WP7, there have been discussions with all pilots to further clarify their needs and their proposed solutions, as the pilots are getting in more technical details in the work that is being currently done in WP7. Finally, T3.2 is at the lower layer of the Reference Architecture which follows the guidelines of the DBVA, and therefore, its output will be taken into account by the system components that are located in the upper layers, and more precisely the semantic interoperable engine in WP4, and the analytical tools that will be developed in the scope of WP5.

1.3. Structure

This document is structured as follows: Section 1 introduces the document, putting the work reported in this deliverable under the context of the project, highlighting its relation with other tasks of the

DoA. Section 2 provides a state-of-the-art analysis of existing solutions and frameworks in the wider technological and scientific area of *polystores.* Section 3 introduces the INFINITECH Common Query Language that will be the basis for accessing data across different data stores, while Section 4 presents the overall design of the solution. Section 5 provides the details on the initial implementation that has been provided as a validation of the architecture of this component, targeting one external datastore that is compatible with JDBC. Finally, section 6 concludes the document.

2. State-of-the-Art Analysis on Polystores

Accessing data from diverse and heterogeneous data sources has been long studied and various solutions have been proposed, usually called multidatabase or data integration systems [1][2]. The most typical approach that can be found in a variety of those proposals involves the definition of a common data model and query language that can be used to transparently access data sources via the mediator-wrapper paradigm, which aims to hide the details of the diverse data connectivity and distribution. In the latest years, with the emerge of cloud databases and big data processing frameworks, the multidatabase solutions evolved to what we call nowadays as polystores systems were defined. The latter enable integrated access to traditional relational database management systems, NoSQL solutions and Hadoop data lakes via a common query engine.

A first category of polystores can be considered as loosely-coupled, and resembles much to the traditional multidatabase systems that can deal with autonomous datastores, being accessed via a common interface, following the mediator-wrapper paradigm, where the access is being granted via the common interface exposed by the mediator, while the wrapper implements the details on how to connect and access and retrieve data from the source. Most of the loosely-coupled systems can only support read-only operations. In this category, BigIntegrator [1] integrates data from cloud-based NoSQL big data stores, such as Google's Bigtable, and relational databases using its own query language, which however does not support Hadoop data lakes, while QoX [3] integrates data from RDBMS and HDFS data stores. SQL++ [4] mediates SQL and NoSQL data sources through a semi-structured common data model. Its query engine is capable of translating the subqueries of a statement to native queries that will be executed in the target datastores. BigDAWG [5][6] on the other hand, instead of translating the different datastore into a common data model, it defines the *islands of information*, where its island corresponds to a specific data model and its language and provides transparent access to the target database. On top of that, it enables cross-island query execution by exchanging and moving intermediate results between the different islands.

Another category of polystores that has an opposite approach, includes what are to be considered as tightly-coupled polystores. Their goal is to integrate Hadoop or Spark for big data analysis with traditional relational database management systems. They tend to trade autonomy for performance, and they provide massive parallelism, using shared-nothing nodes in a cluster, and can benefit by the use of high-performance computing. One example is Odyssey [7] which enables storing and querying data within HDFS and RDBMS, using opportunistic materialized views. MISO [8] aims to tune the physical design of a multistore system with the aim of improving the overall performance when used in big data processing. JEN [9] aims to join data coming from two different datastores such as Hadoop and traditional relational database management systems, parallelising join algorithms and minimizing data movement when executing these algorithms. Polybase [10] enables HDFS access using SQL scripting language. Moreover HadoopDB [11] provides MapReduce access to several RDBMS systems that are supported, it establishes data connectivity and then executes SQL queries that return key-value pairs to be further processed by the MapReduce. Teradata IntelliSphere [12] provides an integrated data access over heterogeneous data sources, that are SQL compatible though.

Apart from these two categories of polystore systems, in the latest years, hybrid solutions have been proposed in the literature. These hybrid solutions have evolved and are widely used by the industry. On the one hand, they support data source autonomy as loosely-coupled systems do, while on the other hand, they preserve parallelism by exploiting the local datastores, as in tightly-coupled systems. They can be seen as parallel query engines that provide several different connectors to external sources that can be executed in a parallel fashion as well. One representative of this hybrid category is Spark SQL [13] which is a parallel SQL engine that provides tight integration between relational and procedural processing via a declarative API and takes advantage of massive parallelism when executing the query statements. It defines the notion of DataFrames that map arbitrary object collections that are being retrieved from the local datastores into relations, and thus, it enables relational operations. Presto [14] is a distributed SQL query engine running on a cluster of machines and can process analytical queries against big data sources via massively parallel processing. This is achieved by using a coordinator process, and multiple workers that make use of connectors that provide the interface with the external data sources and provide all kind of metadata information to the coordinator to optimize the query execution. Apache Drill [15] is also a distributed query engine for large-scale datasets that makes use of massive parallel processing. *Drillbit* services run at each node and receive the query, compile it to an optimized execution plan and exploit data locality to further extend the level of parallelism. Myria [16] is yet another recent polystore, built on a shared-nothing parallel architecture, which efficiently federates data across diverse data models and query languages. Finally, Impala [17] is an open-source SQL engine with massive parallelism capabilities, operating over Hadoop data processing environment. As opposed to typical batch processing frameworks for Hadoop, Impala provides low latency and high concurrency for analytical queries.

In INFINITECH, we provide the *Integrated Polystore Engine,* a framework to seamlessly retrieve data stored in heterogeneous datastores via a common query language, whose main difference with the proposed solutions is that not only does it enable parallel integration with external data sources, but it also combines massive parallelism with native queries that enables the exploitation of the unique characteristics of the target data management system. Moreover, all aforementioned solutions either retrieve the majority of the data in an intermediate layer or process there the query in a parallelized fashion, or push down the query execution to the node, which implies a lot of data movement when joining data sources from different stores. One innovative aspect of the INFINITECH Integrated Polystore Engine is its ability to efficiently execute join operations using *bind joins.* A query statement can be analysed in subqueries, each of which is targeting an external datastore. These are handled by table functions, which can be considered as query operators that can be considered by the query planner and query optimizer of the central data management layer of INFINITECH. Once the integration of the Polystore engine of the platform with the data management is achieved, then bind joins can be proposed automatically when selecting the optimal query execution plan and data execution over integrated heterogeneous datastores can be more effective. Combining this aspect with the ability to express a subquery by native language or scripts allows to fully exploit the power and unique characteristics of the target data management system, as opposed to static mappings to a common data model used to execute query statements across datastores.

3. INFINITECH Common Query Language

As mentioned in the previous section, the Integrated Polystore Engine component of the platform fits into the hybrid category of polystores, being both loosely-coupled, thus providing autonomy on the external datastores and at the same time tightly-coupled, providing a mechanism for massive parallelism, having a common data model. The main difference with the aforementioned solutions is that the data model is not static but instead, it can be created dynamically fitting to the needs of the submitted query and the target datastores. To do this, we need a novel query language that can allow this. Towards this direction, we introduce the INFINITECH Common Query Language that will be based on the CloudMdsQL [18] and its grammar specification can be found in the appendix of this document. The latter is an SQL-based scripting language with extended capabilities that allows embedding subqueries to be expressed in terms of each data store's native query interface. The common data model respectively is table-based, with support of rich datatypes that can capture a wide range of the underlying data stores' datatypes, such as arrays and JSON objects, as a way of handling non-flat and nested data, with basic operators over such composite datatypes.

3.1 Query Language

In this subsection, the design and the basic principles of the INFINITECH Common Query Language will be presented. The latter requires a deep expertise and knowledge by the data scientists and application developers regarding the specifics of the underlying datastores to exploit their unique characteristics, as well as awareness about how data are organized across them. It is important to highlight that the Integrated Polystore Engine component takes into account only read-only operations, allowing the data ingestion and modification to happen at the data store level. As a result, the integrated queries will make use of *projection* over several *selections* that consist of native subqueries. On the component level, the results of the *selections* are being *joined* and the *projection* is being applied. At the level of the scripting language, that will involve a SELECT statement over native subqueries. The latter are defined as *table functions*, also called *named table expressions*. This means that a *function/expression* is being submitted that returns a virtual table, which has a name and a signature, consisting of the names and types of the columns of that virtual table. The *function/expression* can be either a regular SQL SELECT statement or a native statement expressed in the query interface of the target datastore. It is important to notice that the Integrated Polystore Engine component includes a query compiler, which can analyse the SQL statement and re-write it in order for the engine to execute it more efficiently. To highlight the difference between SQL and native query, let's assume that we have an integrated query targeting a traditional relational SQL compatible database and MongoDB[4], which is a document-based datastore with its own query interface. The integrated query expressed in the INFINITECH Common Query Language will need to join two subqueries, one expressed in standard SQL and the other in a native way. The query will be the following:

```
T1(x int, y int)@rdb = (SELECT x, y FROM A)
T2(x int, z array)@mongo = {*
  return db.A.find( {x: {$lt: 10}}, {x:1, z:1} );
*}
SELECT T1.x, T2.z FROM T1, T2
WHERE T1.x = T2.x AND T1.y <= 3
```

[4] https://www.mongodb.com/

In this example, we define two named table expressions, T1 and T2 which target datastore rdb (an SQL compatible database) and datastore mongo (a MongoDB database). T1 is defined by standard SQL while T2 holds a native expression (it is included in the bracket symbols {* *}). The result will be the *projection* of T1.x and T2.z from the *join* operation of the results of those two subqueries that will be sent independently to the two target datastores. At this point, we need to highlight that the query includes a *filter* condition (T1.y <= 3) that is applied on T1, which holds a standard SQL statement. The query compiler of the Integrated Polystore Engine component can identify an optimization and can push down this filtering in the T1 itself, in order for the results of the latter to be fewer, as the filtering will be applied at the external datastore level.

Apart from being able to send native or SQL queries, the INFINITECH Common Query Language also allows table functions to be defined as expressions in a scripting language (i.e. Python, JavaScript). This is useful when datastores offer only API-based query interface and do not simply allow connections where the user submits a query statement. These scripting expressions can either return and put the returned tuples into a table-like result set or return an *iterable* object representing the result set, as the example above with MongoDB). Moreover, the INFINITECH Common Query Language can use as input the result of other subqueries, thus allowing for nested queries. Last but not least, in the same way it is possible for traditional data management systems to create views or stored procedures, our language allows to create *named expressions* via the corresponding command. These expressions are defined during the execution of this command and stored in the global catalogue and can be later re-used and referenced by other queries. This can be of great importance for the data analysts who do not understand the deep details and unique characteristics of each of the underlying datastores, but they are rather interested in executing statements and making their analysis on the results. The data scientist or administrator who fully understands the underlying data technologies and the specifics of the data organization can prepare those named expressions to be frequently re-used.

3.2 Bind Join

As we have mentioned in section 2, one of the problems of the polystore systems is their inefficient implementations of *join* operations in case they need to combine data coming from several data sources. They either retrieve all intermediate results of the involved subqueries in memory or apply the operation using massive parallelism, or they tend to move the datasets across the data stores so as to perform the operation at the lower level, without avoiding the movement of the large amount of data containing the majority of the intermediate results. Instead, the Integrated Polystore Engine component of INFINITECH makes extensive use of *bind joins*, a technique firstly proposed by IBM that has been later thoroughly described in the literature [19]. It allows performing efficient semi-joins across datastores, by re-writing the subquery statement with the aim of pushing down the join conditions. This means that the distinct values of the attribute(s) that are involved in the join condition that are retrieved by the left operator (the subquery that is on the left side of the *join* clause) are passed as a filter to the right operator, through an *in* clause. The following example aims to clarify this:

```
A(id int, x int)@DB1 = (SELECT a.id, a.x FROM a)
B(id int, y int)@DB2 = (SELECT b.id, b.y FROM b)
SELECT a.x, b.y FROM b JOIN a ON b.id = a.id
```

In this example, we have two subqueries projecting their ids and values from a selection among two tables (one per subquery), and an *equity join* operation on those subqueries over their ids. The query optimizer will make use of the *bind join* technique and will link the results of the left operator to the right-hand side of the operation. During the query execution, the relation B will be retrieved from the DB2 datastore using its own query mechanism. Then, the Integrated Polystore Engine component will make use of the list of the distinct values of B.id and will push them into an *in* operation down to DB1 to filter the values of the selection of the table A. Assuming the list of values of B.id are [b1, b2 …. bn], the query A will be transformed as follows:

```
SELECT a.id, a.x FROM a WHERE a.id IN (b1, …, bn)
```

In case of native queries, the compiler cannot re-write the query in the native subquery, and therefore, the data scientist or application developer has to explicitly declare its use in the script, making use of a *JOINED ON* clause in the signature of the named table, as the example below that involves a native script compliant with the MongoDB interface.

```
A(id int, x int JOINED ON id
    REFERENCING OUTER AS b_keys)@mongo =
{*  return db.A.find( {id: {$in: b_keys}} );  *}
```

In this example, the *in clause* of the native statement will take as parameters the values of the outer intermediate result that is being passed into the *b_keys* reference. By doing this, the query executor is enforced to submit firstly the subquery A in order to retrieve the values to be further used in the reference parameter, and then provides this reference to the native query.

Even if the bind join technique seems very promising and reduces a lot the execution time and data movement, thus making the execution of the query much more efficient, is not always a panacea and is restricted by several causes which can create an important overhead. Firstly, the bind join operation demands that the query executor waits for the completion of the execution of one of the two operators, in order to retrieve all required values, before pushing them down for filtering out the results of the second operator. Secondly, if the number of distinct values of the join attribute that will be pushed down is large, the bind join operation may slower down the performance as it will require to send a lot of data via the network, while the selectivity of the data on the second operator will not be adequate to filter out values, and as a result, neither the query execution time on the second subquery will be reduced, nor the amount of retrieved data will be lowered. Due to this, it is important for both external datastores to expose a cost model in order for the query optimizer to predict the number of rows and distinct values that each of the subqueries is expected to return. However, if a native query is involved (thus, the query compiler cannot understand what this is about, as it sees it as a black box) or the cost information is not available, the query can still take this decision, but on the runtime: it will attempt to perform a bind join, it will start collecting the intermediate results of one of the two operators, and if the number of the distinct join keys exceeds a certain threshold, then execution will fall back to an ordinary *hash join* (as bind joins can be used over *equity joins*). In any case, the data scientist and the application developer have the best of the knowledge of the data distribution and can explicitly request the execution of a bind join by using the reserved word BIND in the statement (i.e. FROM b BIND JOIN a).

3.3 MFR Extensions

To address distributed processing frameworks (such as Apache Spark) as data stores, the INFINITECH Common Query Language introduces a formal notation that enables the ad-hoc usage of user-defined map/filter/reduce (MFR) operators as subqueries to request data processing in an underlying big data processing framework (DPF) [20]. An MFR statement represents a sequence of MFR operations on datasets. In terms of Apache Spark, a dataset corresponds to an RDD (Resilient Distributed Dataset – the basic programming unit of Spark). Each of the three major MFR operations (MAP, FILTER and REDUCE) takes as input a dataset and produces another dataset by performing the corresponding transformation. Therefore, for each operation there should be specified the transformation that needs to be applied on tuples from the input dataset to produce the output tuples. Normally, a transformation is expressed with an SQL-like expression that involves special variables; however, more specific transformations may be defined through the use of lambda functions. Let us consider the following simple example inspired by the popular MapReduce tutorial application “word count”. We assume that the input dataset for the MFR statement is a text file containing a list of words. To count the words that contain the string ‘cloud’, we write the following composition of MFR operations:

```
T4(word string, count int)@spark = {*
   SCAN(TEXT, 'words.txt')
  .MAP(KEY, 1)
  .REDUCE(SUM)
  .FILTER( KEY LIKE '%cloud%' )
*}
```

For defining map and filter expressions, the special variable TUPLE can be used, which refers to the entire tuple. The variables KEY and VALUE are thus simply aliases to TUPLE[0] and TUPLE[1] respectively.

To optimize this MFR subquery, the sequence is subject to re-writing according to rules based on the algebraic properties of the MFR operators, as explained in [20]. In the example above, since the FILTER predicate involves only the KEY, it can be swapped with the REDUCE, thus allowing the filter to be applied earlier as a way to avoid unnecessary and expensive computation. The same rules apply for any pushed down predicates, including bind join conditions.

4. Design of Integrated (Polyglot) Persistence

This section will illustrate the basic architectural principles that the Integrated Polystore Engine component of INFINITECH follows and will focus on the details of the overall design of the proposed solution. Moreover, it will give some technical insights on how the integrated query processing can be executed in parallel and how the high expressivity of the INFINITECH Common Query Language fits together with the design of the implementation.

4.1 Basic Architectural Principal: The mediator/wrapper paradigm

During the state-of-the-art analysis in the area of the polystore systems, it was clear that most of the proposed solutions rely on the mediator/wrapper paradigm, or modifications of it, which tends to be widely used when there is the need to access data and information coming from several external data sources that are both diverse and heterogeneous. This heterogeneity is highlighted by the fact that they not only provide different means and protocols for connectivity, exposing different APIs and making use of different types of drivers required to establish a connection, but they are also compatible with different query scripting languages. To make things worse, each of the data store may have its own data model to store data and return results (i.e. a relational model in traditional relational database management systems and a JSON schema in document-based datastores), and according to the data source, data might be structured, semi-structured or totally unstructured. Due to this, it is very complex to implement a unique way to execute query statements and access integrated data spanned among different stores.

To facilitate the implementation of a polystore, the mediator/wrapper paradigm is introduced aiming to hide the implementation details of the data connectivity and data access at a lower level, providing generic interfaces that the query engine can make use in order for the execution of the query to become transparent regardless the target datastore. Following this approach, a central building block in terms of a component diagram (i.e. it does not have to be a centralized component, it might rather be implanted in a distributed manner) is the *mediator.* The latter is responsible of the orchestration of the execution of the query statement. To access data and retrieve the results, it makes use of the *wrappers*, which hide the complexity and the internal details of how to access data and retrieve the intermediate results. Wrappers themselves can also be implemented in a distributed manner, thus allowing for intra-operation parallelism, if we consider that the data retrieval from an external datastore is a single operation in the query execution tree, no matter the complexity of that operator.

Each *wrapper* is responsible to handle a specific type of a datastore. The polystore supports as many different types of external datastores as the wrapper implementations for those datastores are available and supported by the former. The *wrapper* provides the following functionalities:

- It retrieves a subquery in a predefined format (i.e. it can be a standard SQL statement, a native query, or an agreed model of the subquery in a structured way).
- If not a native query, it translates the input to a query that is compatible and equivalent to the supported dialect and query interface that the target datastore accepts.
- It can establish a connection to the target datastore. Internally, it makes use of a *data connector* subcomponent that holds the specific driver to the target datastore, and has implemented all the details on how to open, maintain, close a connection, send statements and receive results over that connection according to the protocol with which the datastore is compatible. The internal details are irrelevant from the wrapper point of view, so it might

use a pool of connections and do whatever type of optimization it considers necessary, as long as it does not violate the agreed protocol with the datastore.

- It is able to retrieve data over that connection, iterate over the return result and close the connection (via the interfaces exposed by the *data connector*) once the data has been returned.
- It is responsible for transforming the data coming from the target datastore into the common data model that the whole solution is using, and for returning the results back. The retrieved data are being transformed to tuples yield to a *named table expression* that the core of the Integrated Polystore Engine makes use of.
- (Optional) It may implement a cost estimation model. A requirement for this to happen is the ability of the wrapper to request statistics from the target datastore, which is possible only if the latter is able to expose this kind of information. This includes types of indexes, number of rows per table, information about the histogram of the tuples over the indexes, number of hits, history of submitted query statements, etc.
- (Optional) It may be able to transform the incoming query, explore different query execution plans by applying various transformation rules, and re-write the query to an equivalent one that can improve the overall execution. In order for this to be possible, it is required that the *wrapper* can perform a cost estimation model that would be taken into account by its query optimizer when investigating the cost of the explored query execution plans.

On the other hand, the *mediator,* as the central building block in this architectural paradigm orchestrates the whole query execution of the integrated statement and makes use of the available *wrappers.* It receives as an input the submitted integrated statement that consists of one or more subqueries, each of which is targeting a specific external datastore. Internally, it contains a query compiler that is capable of transforming the script into a structured query plan that can be parsed by the query planner. The latter explores and suggests alternative and equivalent query plans for execution. For instance, in case that a subquery is a standard SQL statement, it can be decided to push some *filter* operations down to the external datastore engine to reduce the amount of data that will be transmitted across the network and the amount of memory that will be required to process the intermediate result. In case that the external datastore can provide statistics that can be taken into account by the cost estimation model of the mediator, then further improvements can be decided on the preparation phase of the execution, as it was highlighted in the case of the *bind joins* in section 3. At the end of the preparation phase, the query plan to be executed is decided. It consists of the subqueries that need to be executed in the target datastores. The *mediator* then sends these queries into the corresponding wrapper and retrieves the results in the form of a *named table expression,* or a *virtual table.* To do so, it provides a common interface that all *wrappers* must implement, so that the overall execution will be transparent from the *mediator* point of view. As a result, it uses this interface to submit the corresponding subquery and retrieves the result in the agreed format, hiding all the complexities for data connectivity and data access to the *wrapper,* following the *separation of concerns* concept that is of major importance in the development of system and software solutions. Finally, it merges the results and returns the result set back to the data scientist or application developer.

The bird-eye-view of this architecture principles that the INFINITECH Integrated Polystore Engine component follows is depicted in Figure 1.

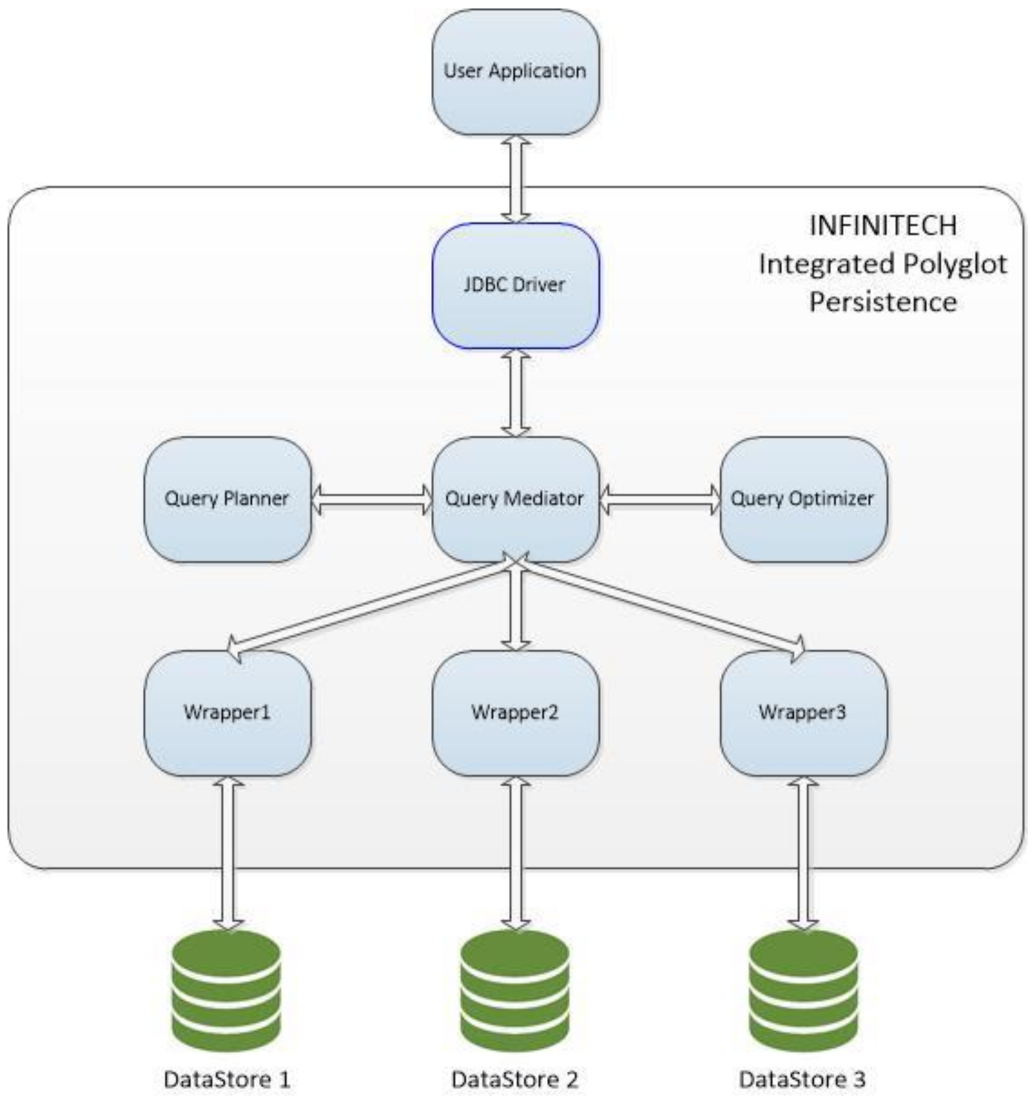

Figure 1: The Mediator/Wrapper paradigm in INFINITECH Polystore Engine

The data scientist or application developer submits an integrated query in the system for execution. This is being done in a seamless way, using a unique interface which is the standard JDBC. The integrated query arrives to the *mediator* component that starts the preparation phase. As explained previously, it compiles the query and transforms it in a structured way for further processing and it communicates with the *planner* that explores the space of possible query transformations into equivalent ones. Then it communicates with the *optimizer* which takes the responsibility to estimate the most cost effective one, taking into account its own cost estimation models (i.e. pushing down a *filter* and *projection* operations before the *selection* is always more efficient than applying those operations after the data has been retrieved from the datastore). When the *optimizer* decides about the plan to be executed, then the *mediator* sends the several subqueries down to the corresponding *wrappers* by invoking the common interface that each of the wrapper implements. At that point, it is totally transparent to the *mediator* how the data will be retrieved, as this logic is hidden by the *wrappers* themselves. They are responsible to establish the connection to the target datastore, send the query for execution, retrieve the results and return them back to the *mediator* in a form of a *virtual table.* The *latter* receives the intermediate results, applies the *join operation* and returns the data in the form of a *ResultSet* back to the user via the JDBC connection.

According to the capabilities of each of the target datastores, the intermediate result might be returned as a whole, where the execution of the subquery has been finished, or might be returned via batches. In the first case, the *mediator* has to wait for the execution to be completed before continuing on the execution of the operation that is placed in the upper node of the query tree. This is not efficient enough, as it will block the overall execution waiting for this operation (the intermediate data retrieval through the wrapper) to be finished first. If the external datastore is capable of returning the results in batches, the overall execution can be much more accelerated, as the data pipeline that is being established by the *mediator* can be executed on the fly, as data are received, without the need to block the execution and have to maintain all data in memory. Taking into account that the *mediator* has to perform one of the most expensive operations, the *join*, this can be of great benefit: This will allow the execution of *merge joins* in case the join attributes are ordered over an index, and will also improve the execution of *nested-loop joins* and *hash joins,* as the nested loop will be sent to the right-hand operator the moment the data arrives from the left-hand, while the *hash* in the hash join can release resources much more easily.

It is important to highlight at this point the fact that the *mediator* along with the *planner* and *optimizer* are internal parts of the INFINITECH central data repository, with the appropriate extensions that are planned in order to take into account the additional requirements that the polyglot engine introduces. As a result, the INFINITECH Integrated Polystore Engine component does not have to re-create the *join* operations, but instead, it can rely on the core of the query engine of the data repository. As explained in more details in the corresponding deliverables of T3.1 “Framework for Seamless Data Management and HTAP” (D3.1, D3.2 and D3.3), its query engine provides massive parallelism processing and provides intra-query and intra-operation parallelism, making the execution of these types of operations much more efficient. The scope of this deliverable is to provide technical information on the polyglot extensions and not to give more insights on the query engine as a whole, which is part of T3.1.

4.2 Parallel Integrated Processing using the INFINITECH Common Query Language

As it has been analysed in D3.1, the distributed query engine of the INFINITECH central data repository is designed to be integrated with arbitrary data management clusters, which can store data in their natural format, without the need of pre-processing in order to be transformed to a compatible data schema, and can be retrieved in a parallel fashion by either executing declarative queries or by running specific scripts. The query engine supports a variety of diverse data management clusters, from distributed raw data files, parallel SQL database management systems, sharded NoSQL databases and parallel processing frameworks such as Apache Spark. To that sense, the query engine of the data repository of the platform can be integrated with the Integrated Polystore Engine component and thus transform it into a powerful *big data lake* polyglot engine that is capable of taking the full advantage of both expressive scripting and massive parallelism. Moreover, as it was described in the previous subsections, the integral query engine of the INFINITECH data repository supports the efficient execution of a variety of implementation of the *join* operator, and by providing intra-operator parallelism; these operations can be executed in a distributed manner in parallel. As a result, joining data coming from native datasets being stored in external datastores along with the internal data tables of the repository itself can be applied thanks to the most suitable implementation that will exploit efficient parallelism. To highlight that fact, we will illustrate in the following examples how the execution of a parallel join operation across a relation table and the result of a JavaScript subquery to a document-based datastore, like MongoDB, is being done along with the join of a data table with a MFR query, using the INFINITECH Common Query Language and the Integrated Polystore Engine.

Let's assume that we have a table (collection in MongoDB terminology) called *orders* with the following data schema:

```
{order_id: 1, customer: "ACME", status: "O",
 items: [
  {type: "book", title: "Book1", author: "A.Z.",
          keywords: ["data", "query", "cloud"]},
  {type: "phone", brand: "Samsung", os: "Android"}
] }, ...
```

This is an example of semi-structure data as each of the records in the *orders* contains an array of *items*, whose attributes differ according to the type. We need to return the title and the author of all books in the *orders* by a given customer. This can be expressed with a *flatMap* operation in JavaScript and a MongoDB *find()* operation. This can be depicted in the following code listing, where this expression is being wrapped into a *named table expression* of the INFINITECH Common Query Language:

```
BookOrders(title string, author string,
           keywords string[])@mongo =
{*
  return db.orders.find({customer: "ACME"})
  .flatMap( function(v) {
    var r = [];
    v.items.forEach( function(i){
      if (i.type == "book")
        r.push({title:i.title, author:i.author,
                keywords:i.keywords});
    } );
    return r; });
*}
```

We can see from the above code listing that we define a *virtual table* called *BookOrders* with the specified signature (title column as a String, author column as a String etc.) which will be executed in the *@mongo* external datastore, using a native script (as indicated by the {* *} brackets) where we place the *flatMap* inside the *find()* MongodB operation. Furthermore, we need to join the result of this with a table named *authors* that is being stored inside the central data repository. The integrated statement according to the INFINITECH Common Query Language will be the following:

```
SELECT B.title, B.author, A.nationality
FROM BookOrders B, Authors A
WHERE B.author = A.name
```

This involves an *equity join* operation over a *scan* operation on the right-hand and a polyglot operation on the left. Let's assume for simplicity that the query optimizer decides to use a *nested-loop join* operation. For what concerns the scan operation over the internal data table, this can be executed in parallel due to the distributed data management clustering that supports this. Regarding the polyglot operation, this is related to whether the wrapper can be executed in parallel or not. In any case, the *nested-loop join* operation also supports intra-operator parallelism. This means, that the data pipeline of the integrated query plan can be configured and once data are retrieved by the polyglot operation, then the *nested loop* implementation of the *join,* can use the hashcode of the join attribute and send the tuple to the related *worker* to execute the right-hand of the *join,* as explained in more details in deliverable D3.1.

Additionally, a more sophisticated data transformation logic needs to be applied over the unstructured data before being able to be processed by relational operators. In the following example, we need to analyse the logs of a scientific forum to identify the top experts for particular keywords, assuming that the most influencing user for a given keyword is the one who mentions the keyword most frequently in their posts. We assume that the application keeps the log data in the non-tabular structure that is depicted below:

```
2014-12-13, http://..., alice, storage, cloud
2014-12-22, http://..., bob, cloud, virtual, app
2014-12-24, http://..., alice, cloud
```

There are text files where a single record corresponds to one post which contains a fixed number of fields about the post itself (timestamp, link to the post, and username in the example) followed by a variable number of fields storing the keywords mentioned in the post. The unstructured data needs to be transformed into the following tabular format:

```
KW        expert   frequency
cloud     alice    2
storage   alice    1
virtual   bob      1
app       bob      1
```

Such transformation requires the use of programming techniques like chaining map/reduce operations that should take place before the data is involved in relational operators. This can be expressed with the following MFR subquery with embedded Scala lambda functions to define custom transformation logic:

```
Experts(kw string, expert string)@spark = {*
  SCAN( TEXT, 'posts.txt', ',' )
  .MAP( tup=> (tup(2), tup.slice(3, tup.length)) )
  .FLAT_MAP( tup=> tup._2.map((_, tup._1)) )
  .MAP( TUPLE, 1 )
  .REDUCE( SUM )
  .MAP( KEY[0], (KEY[1], VALUE) )
  .REDUCE( (a, b) => if (b._2 > a._2) b else a )
  .MAP( KEY, VALUE[0] )
*}
```

Skipping the details of the MFR query, we can see that we define a *named table expression* called *Experts,* which provides its signature and makes use of a native MFR query. We further join this table with the *BookOrders* that were defined earlier and whose data are persistently stored in MongoDB. We can write the integrated query in the following way:

```
SELECT B.title, B.author, E.kw, E.expert
FROM BookOrders B, Experts E
WHERE E.kw IN B.keywords
```

We illustrate in this example how the *bind join* supported by the Integrated Polystore Engine component can be used for optimal execution of this query. In this case, the bind join condition (which involves only the kw column) can be pushed down the MFR sequence as a FILTER operator. As per the MFR re-writing rules, this would take place immediately after the FLAT_MAP operator, thus reducing the amount of data to be processed by the expensive REDUCE operators. To build the bind join condition, the query engine flattens B.keywords and identifies the list of distinct values.

By processing such queries, the distributed query engine of the INFINITECH central repository can take advantage of the expressivity of each local scripting mechanism, enabled via the use of the Common Query Language of the platform, yet allowing for results of subqueries to be handled in parallel by the query engine itself and be involved in operators that utilize the intra-query parallelism. The query engine architecture is therefore extended by the implementation of the Integrated Polystore Engine to access in parallel shards of the external data store through the use of DataLake distributed wrappers that hide the complexity of the underlying data stores' query/scripting languages and encapsulate their interfaces under a common DataLake API to be interfaced by the query engine.

5. Implementation of the Integrated (Polyglot) Persistence

This section illustrates the initial implementation of the Integrated Polystore Engine component, giving some high-level overview of how the code is organized along with code examples in the form of pseudo code. As it has been already mentioned, at this phase of the project, a wrapper has already been implemented, which is capable of accessing data that is stored in a traditional relational database management system. The purposes of this section are to focus on the implementation related to the Integrated Polystore Engine component, and not to give more details on the overall query engine of the INFINITECH central depository. For that, a class diagram regarding how the code of the *wrapper* has been organized is provided, giving analytical details of the functionalities provided by the code, along with a more detailed explanation provided with a pseudo code, that can be used as a roadmap for the implementation of the additional *wrappers* that need to be implemented in order to access code that is stored in other types of datastores.

5.1 Abstract Wrapper Implementation

To facilitate the implementation of wrappers for the corresponding external datastores, various interfaces have been defined that can be used by the *mediator* component, which is part of the query engine of the central data repository. As already explained, having generic interfaces enables the *mediator* to handle all different types of external datastores, in a unified manner. To test our design, we implemented a *wrapper* as a proof-of-concept, and we assume that it can be used to establish connections and access data in a traditional SQL-compatible relational datastore. We call our mock datastore as *rdb.* The class diagram is depicted in Figure 2.

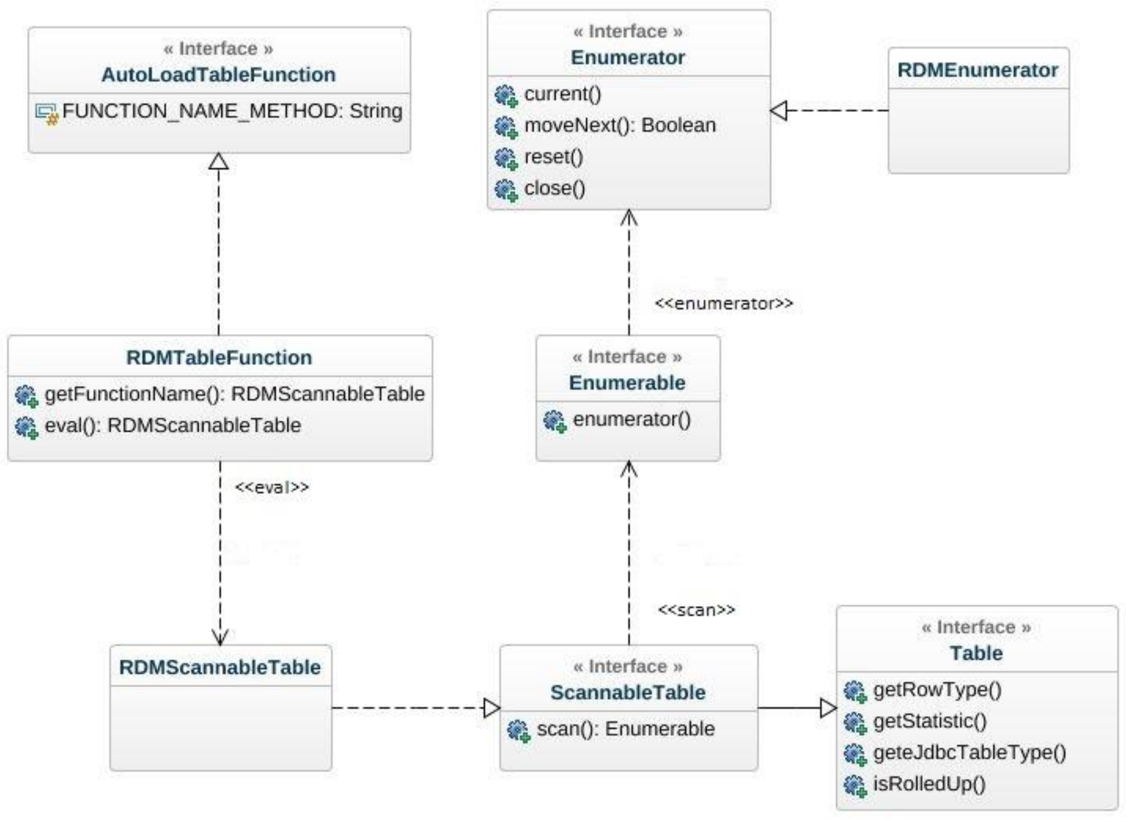

Figure 2: RDBWrapper class diagram

From the figure, we can see that each wrapper must implement the *AutoLoadTableFunction.* As it has already been mentioned, a subquery is being submitted either in declarative language, or via a scripting expression, and it formulates the *table function.* For our implementation of the *rdm* wrapper, the *RDMTableFunction* class implements this interface. Upon initialization of the query engine, this class is loaded via Java Reflection. It implements two methods: *getFunctionName* which returns the name of the wrapper, which in our case will be '*rdb'*, and the *eval* which returns a *ScannableTable* object, which implements the business logic that the wrapper is actually doing. To make things clearer, let's assume we have the following statement in the INFINITECH Common Query Language:

```
A(id int, x int)@RDB = (SELECT a.id, a.x FROM a)
```

This illustrates a subquery that will be executed in the *@RDB* wrapper. In that case, the query engine holds a map with all the wrappers, along with their names, so this code will be passed to the *RDMTableFunction* to be executed. The *eval* method expects two string arguments: the subquery to be executed and the signature of the result. In this example, the query will be the (SELECT a.id, a.x FROM a) statement, and the signature the (id int, x int).

The query engine invokes the *eval* method to get the results, and the latter returns an implementation of the *ScannableTable.* The latter extends the *Table* interface, and it will be used by the query engine as it consists of the polyglot operation that is part of the data pipeline established by connecting the various operations needed for the retrieve data according to the tree of the query plan. It defines various methods that are relevant to the query engine itself for creating the list of equivalent query trees and for providing information to the query optimizer, along with information needed for the real-time execution of the operator. The functionality and rationale of these methods are beyond the scope of this deliverable, apart from the *scan.* This is used by the query engine to retrieve the *Enumerable* object that will return the raw data transformed to the common model, as retrieved by the external datastore.

The *Enumerable* object returns an implementation of the *Enumerator* interface, which implements the methods that actually access data in the external datastore. As an *enumerator*, it defines methods for closing the object, which will imply to close the connection to the external database, check if there are more data to be returned, reset the pointer to the first tuple of the retrieved data, and to actually return the tuple, via the *current* tuple. In our implementation, the *RDMEnumerator* implements this logic, and the next subsection goes deeper into the details, with pseudo code snippets.

5.2 RDM Wrapper Implementation details

From the previous subsection, the main interfaces that the developer of a wrapper should provide to the INFINITECH Integrated Polystore Engine component to grant access to an external datastore are the following:

- AutoLoadTableFunction
- RDMEnumerator

All other interfaces are being provided by the core of the query engine of the INFINITECH central depository and its polyglot extensions. Regarding the *AutoLoadTableFunction,* this interface defines two methods. The first one, returns the name of the wrapper so that the query engine can be informed that there is an additional polyglot operator to be taken into account and might be addressed by a subquery of an integrated statement. Thus, the following code illustrates its implementation:

```
public static String getFunctionName(){
        return "rdb";
}
```

We have named our wrapper as *rdb*, in the sense that it is targeting an external relational database management system, therefore, this term will be used in the integrated statement to drive the component to use this implementation when addressing subqueries with the *@rdb* indicator.

Moreover, the *eval* method returns an implementation of the *ScallableTable* interface, which extends others that are necessary from the query engine to manipulate these types of objects. A pseudo code snippet for implementing this method can be the following:

```
public ScannableTable eval(final String query, final String schemaDefinition) {
         //parse the schemaDefinition to grab the field names and types
        String [] fieldNames = getFieldNamesFromSignature(schemaDefinition);
        SqlTypeName [] fieldTypes = getFieldTypesFromSignature(schemaDefinition);

         return new ScannableTable() {
                @Override
                public Enumerable<Object[]> scan(DataContext dc) {
                        return new AbstractEnumerable<Object[]>() {
                              @Override
                              public Enumerator<Object[]> enumerator() {
                                        return new RDMEnumerator(query, fieldNames, fieldTypes);
                              }
                      };
                }

                @Override
                public RelDataType getRowType(RelDataTypeFactory relDataTypeFactory) {
                        return getRowType(relDataTypeFactory, fieldTypes);
                }

                @Override
                public Statistic getStatistic() {
                        return Statistics.UNKNOWN;
                }

                @Override
                public Schema.TableType getJdbcTableType() {
                        return Schema.TableType.TABLE;
                }

                @Override
                public boolean isRolledUp(String string) {
                        return false;
                }

                @Override
                public boolean rolledUpColumnValidInsideAgg(String string, SqlCall sc, SqlNode
sn, CalciteConnectionConfig ccc) {
                        return false;
                }
         };
  }
```

In this pseudo code, it is depicted that a new instance of the *ScallableTable* interface is being created and its methods are overridden by the implantation needed by this specific wrapper. It illustrates that at the beginning, the String containing the schema definition is being parsed because this information will be needed during the scanning phase to transform the data to the common model that has been defined by the integrated statement. Then, the *scan* method creates a new *Enumerator* object, which is the *RDMEnumerator* and which has been provided for this wrapper. It is important to be highlighted at that point that custom enumerator defines a constructor whose arguments must be the column names and types of the signature. By doing that, the instantiation object of this class will have all this information available so as to proceed for the proper transformation of the retrieved data.

Regarding the *RDMEnumerator* class itself, there are several methods that need to be implemented by the interface. It has been introduced an additional one with *private* visibility which manages to open and establish a connection with the target datastore.

```
private void init() {
        if(it is already opened) {
                return;
        }

        try {
                this.connection = createConnection(connection arguments)
                this.stametement = this.connection.createStatement();
                this.resulSet = this.stametement.executeQuery(query);
        } catch(SQLException | SAFFederatorException | CQEException ex) {
                throw new RuntimeException(ex);
        }

        setAlreadyOpen();
}
```

This method checks if the connection to the external datastore is already open, and if not, it establishes a connection, creates the statement object according to the JDBC standard, and executes the query, storing the result set that needs to be parsed later so as to retrieve and transform the data. It is worth to be mentioned that this is pseudo code illustrating a very basic implementation of a relational wrapper which executes the query and waits for the result. The execution of the query might be a blocking operation or not, depending on the type of the query and the implementation of the JDBC driver of the specific datastore. A more sophisticated approach would be to open a new thread that will be responsible to execute this query in parallel with the preparation of the query engine to settle the data pipeline and begin to request data. This thread could retrieve the data in parallel and feed the result in a *blockingqueue* that could be used as the pipeline of this thread and the thread opened by the query engine to execute these lines of code. As a result, the data will be available when the query engine will start fetching them, as the corresponding method would pick them from this *blockingqueue* that would have already been started to be filled with row data from the external datastore.

Regarding the *moveNext* method, it has to check whether or not there are more data to be retrieved from the external datastore. A code snippet could be the following:

```
@Override
public boolean moveNext() {
        init();
        return this.iterator.moveNext();
}
```

It firstly checks if the connection is already open, and if not, it establishes the connectivity. As a result, the very first time that this method will be invoked, the wrapper will open the connection to the external datastore and will execute the query. As we are mocking a traditional relational database management system that provides a JDBC interface, this code relies on its implementation to check this. In a more sophisticated approach with data being retrieved in a parallel thread, this code will have to fetch data from the *blockingqueue* until an *END_FLAG* is received, that will signal the end of the dataset, and the code will be unblocked and return false.

The *current* method returns the current tuple of the retrieved data. A pseudo code could be the following:

```
@Override
public Object[] current() {
        return this.iterator.current();
}
```

As the *RDMEnumerator* mocks a JDBC compliant datastore, this code relies on its implementation to return the current data. It is important to be mentioned here that as both stores, the external one and the polyglot, are relational data stores, there is no need for data transformation. However, in cases of a document-based datastore or a Hadoop Data lake, the code snippet would have to transform the raw data into the corresponding format, before sending back the array of objects to the query engine and the *mediator.*

The *reset* method has to set the pointer in the first row of the retrieved dataset. In our case, this operation is not supported by the JDBC standard, so the code snippet must not allow this, and indeed, it throws a runtime exception as follows:

```
@Override
public void reset() {
        throw new UnsupportedOperationException("Reset operation is not supported by the " +
this.getClass().getSimpleName() + ".");
}
```

Finally, when all data has been retrieved by the query engine, the latter closes the *enumerator*. The implementation of the corresponding method must close all open connections and release all resources that have been reserved for the execution of this subquery.

```
@Override
public void close() {
        try {
                if((this.resultSet!=null)&&(!this. resultSet.isClosed())) {
                        this. resultSet.close();
                }
        } catch(IOException ex) {
                Log.warn("Could not close iterator {}. {}", this.
resultSet.getClass().getSimpleName(), ex);
        }
        try {
                if((this.statement!=null) && (!this.statement.isClosed())) {
                        this.statement.close();
                }
        } catch(SQLException ex) {
                Log.warn("Could not close statement for retrieving tuple. {}", ex);
        }
        try {
                if((this.connection!=null) && (!this. connection.isClosed())) {
                        this. connection.close();
                }
        } catch(SQLException ex) {
                Log.warn("Could not close connection to datastore. {}", ex);
        }
}
```

6. Conclusions

This document reported the work that has been done in the scope of the task T3.2 “Polyglot Persistence over BigData, IoT and Open Data Sources” whose goal is to provide a common and integrated way to access data that is stored in a structured, semi-structured or even unstructured fashion over a variety of heterogeneous data stores, in a unified manner.

Towards this direction, firstly a state-of-the-art analysis has been made on the topic of polystore management systems. This revealed the fact that there are two major categories of polystore systems: loosely-coupled and tightly-coupled ones, each of which is focusing either on the autonomy of the external datastores, or on efficient performance when processing data into a common model, using massive parallelism processing. Apart from those, nowadays a hybrid approach which combines the benefits from both approaches is being used widely. It became obvious that the INFINITECH Integrated Polystore Engine component will make use of the mediator/wrapper architectural paradigm, widely used by the majority of the examined solutions, while it will need to provide even greater expressivity in order not to ignore the unique characteristics of the target databases, as most of the proposed solutions do.

The result of this analysis is the definition of the INFINITECH Common Query Language. This deliverable presents the basic principles of this language, which makes use of SQL language, but additionally gives the possibility to write native queries compatible with the target datastores, in a declarative way or via scripting expressions. An integrated statement written in the INFINITECH language consists of several subqueries that are targeting heterogeneous datastores. The common language abstracts this heterogeneity, while also giving the ability to exploit the datastore unique characteristics. Moreover, the importance of the *bind join* and the way this is supported by the common language has been presented, along with an example on how we can use the latter to access Hadoop data lakes with MFR functions.

As the basis for the Integrated Polystore Engine component has been defined to be the INFINITECH Common query language, the general architecture design of this component has been presented. The component diagram highlighted how polyglot extensions are being designed with the aim of being incorporated within the INFINITECH central data repository, while it was widely presented how this integration allows for the parallel execution of integrated queries across different datastores.

Finally, after having described the overall design of this component, we progressed with the implementation of a *wrapper* that can be used as the first polyglot extension that accesses data from an external relational database management system. More technical details have been provided with the intention of giving the system developers the overview of the design with a class diagram, along with the code snippets to highlight what will be needed to be further implemented for the remaining of the wrappers. This can be used as a guideline for further development.

To conclude, the progress of task T3.2 rhymes with the initial plan: the analysis of the competition has been fulfilled, the definition of the INFINITECH Common Query Language has been delivered, the design of the overall component has been published, while there is already a mock-up implementation that can be used as a reference for the development of the remaining of the wrappers. It is worth to be mentioned that this is the first report of the work to be done in the scope of T3.2, and there will be two more iterations, where more details will be given as the component will be more mature and start to be integrated within the INFINITECH data management layer.

Appendix A: Literature

[1] Z. Minpeng, R. Tore, “Querying combined cloud-based and relational databases”, in Int. Conf. on Cloud and Service Computing (CSC), pp. 330-335 (2011)

[2] T. Özsu, P. Valduriez, Principles of Distributed Database Systems, 4th ed. Springer, 700 pages (2020)

[3] A. Simitsis, K. Wilkinson, M. Castellanos, U. Dayal, “Optimizing analytic data flows for multiple execution engines”, in ACM SIGMOD, pp. 829-840 (2012)

[4] K. W. Ong, Y. Papakonstantinou, and R. Vernoux, “The SQL++ semi-structured data model and query language: a capabilities survey of SQL-on-Hadoop, NoSQL and NewSQL databases”, CoRR, abs/1405.3631 (2014)

[5] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, S. Zdonik, “The BigDAWG polystore system”, SIGMOD Record, vol. 44, no. 2, pp. 11-16 (2015)

[6] V. Gadepally, P. Chen, J. Duggan, A. J. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson, M. Stonebraker, “The BigDawg polystore system and architecture”, in IEEE High Performance Extreme Computing Conference (HPEC), pp. 1-6 (2016)

[7] H. Hacigümüs, J. Sankaranarayanan, J. Tatemura, J. LeFevre, N. Polyzotis, “Odyssey: a multi-store system for evolutionary analytics”, PVLDB, vol. 6, pp. 1180-1181 (2013)

[8] J. LeFevre, J. Sankaranarayanan, H. Hacıgümüs, J. Tatemura, N. Polyzotis, M. Carey, “MISO: souping up big data query processing with a multistore system”, in ACM SIGMOD, pp. 1591-1602 (2014)

[9] T. Yuanyuan, T. Zou, F. Özcan, R. Gonscalves, H. Pirahesh, “Joins for hybrid warehouses: exploiting massive parallelism in hadoop and enterprise data warehouses”, in EDBT/ICDT Conf., pp. 373-384 (2015)

[10] D. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, J. Gramling, “Split query processing in Polybase”, in ACM SIGMOD, pp. 1255-1266 (2013)

[11] A. Abouzeid, K. Badja-Pawlikowski, D. Abadi, A. Silberschatz, A. Rasin, “HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads”, PVLDB, vol. 2, pp. 922-933 (2009)

[12] K. Awada, M. Eltabakh, C. Tang, M. Al-Kateb, S. Nair, G. Au, “Cost Estimation Across Heterogeneous SQL-Based Big Data Infrastructures in Teradata IntelliSphere”, in EDBT, pp. 534-545 (2020)

[13] M. Armbrust, R. Xin, C. Lian, Y. Huai, D. Liu, J. Bradley, X. Meng, T. Kaftan, M. Frank-lin, A. Ghodsi, M. Zaharia, “Spark SQL: relational data processing in Spark”, in ACM SIGMOD, pp. 1383-1394 (2015)

[14] Presto – Distributed Query Engine for Big Data, https://prestodb.io/

[15] Apache Drill – Schema-free SQL Query Engine for Hadoop, NoSQL and Cloud Storage, https://drill.apache.org/

[16] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, D. Moritz, B. Myers, J. Ortiz, D. Suciu, A. Whitaker, S. Xu, “The Myria big data management and analytics system and cloud service”, in Conference on Innovative Data Systems Research (CIDR) (2017)

[17] Apache Impala, http://impala.apache.org/

[18] B. Kolev, P. Valduriez, C. Bondiombouy, R. Jiménez-Peris, R. Pau, J. Pereira, “CloudMd-sQL: querying heterogeneous cloud data stores with a common language”, Distributed and Parallel Databases, vol. 34, pp. 463-503. Springer (2015)

[19] L. Haas, D. Kossmann, E. Wimmers, J. Yang. Optimizing Queries across Diverse Data Sources. Int. Conf. on Very Large Databases (VLDB), pp. 276-285 (1997)

[20] C. Bondiombouy, B. Kolev, O. Levchenko, P. Valduriez, "Multistore big data integration with CloudMdsQL", Transactions on Large-Scale Data and Knowledge-Centered Systems (TLDKS), pp. 48-74. Springer (2016)

Appendix B: Grammar of the INFINITECH Common Query Language

This section formalized the INFINITECH Common Query Language. This grammar formalization uses the following notations:

Tokens:

```
UPPERCASE     keyword
lower_case    nonterminal symbol, left-hand side of exactly one rule
Camel_Case    terminal symbol, described below
'x'           single character symbol
stmt          is the starting nonterminal symbol
```

Special grammar symbols:

```
:=   production rule symbol
|    disjunction
()   subexpression
*    zero or more successive occurrences of the preceding construct
+    one or more successive occurrences of the preceding construct
[]   optional construct
;    end of production rule
```

Terminals:

```
Identifier    matches the regular expression [A-Za-z_][A-Za-z_0-9]*
Type          name or synonym of any CloudMdsQL type
Ext_Code      any text surrounded by native expression brackets
Operation     any of the listed below binary operations
Placeholder   dollar sign followed by Identifier
Value         type constructor of any CloudMdsQL type
Natural       matches the regular expression [1-9][0-9]*
```

Binary operations (in order of precedence):

```
*, /, %
+, -, ||
=, <, >, <=, >=, <>
AND
OR, XOR
```

INFINITECH Common Query Language Grammar:

```
stmt                    := dql_stmt | dml_stmt | ddl_stmt | trans_stmt;

dql_stmt                := named_table_expr* select_stmt;
dml_stmt                := (named_table_expr | named_action_expr)* execute_stmt+;
ddl_stmt                := create_expr_stmt | drop_expr_stmt;
```

```
trans_stmt            := START TRANSACTION | COMMIT | ROLLBACK;

named_table_expr      := table_signature ['@' datastore] table_expression;
table_signature       := table_name '(' arg_list [referencing_clause] [withparams_clause] ')';
table_expression      := Ext_Code | '(' select_stmt ')';

select_stmt           := SELECT target_list [from_clause] [where_clause] [group_clause]
                                [having_clause] [order_clause] [limit_clause];
target_list           := target_col (',' target_col)*;
target_col            := expression [AS col_alias];

from_clause           := FROM from_item (',' from_item)*;
from_item             := (table_ref | sub_select) [[AS] table_alias]
                       | join_expr
                       | '(' from_item ')';
join_expr             := from_item [BIND] JOIN from_item ON expression;

where_clause          := WHERE expression;
group_clause          := GROUP BY expression (',' expression)*;
having_clause         := HAVING expression;
order_clause          := ORDER BY expression [ASC|DESC] (',' expression [ASC|DESC])*;
limit_clause          := LIMIT Natural;

named_action_expr     := action_signature '@' datastore action_expression;
action_signature      := action_name [ '(' [referencing_clause] [withparams_clause] ')' ];
action_expression     := Ext_Code | '(' action_stmt ')';

arg_list              := attr_name Type (',' attr_name Type)*;
name_list             := Identifier (',' Identifier)*;

referencing_clause    := REFERENCING name_list;
withparams_clause     := WITHPARAMS arg_list;

execute_stmt          := EXECUTE '@' datastore action_expression
                       | EXECUTE action_ref
                      ;
insert_stmt           := INSERT INTO table_ref [ '(' column_ref [',' column_ref] ')' ]
                         ( select_stmt | VALUES value_list );
update_stmt           := UPDATE table_ref SET update_col (',' update_col)* [where_clause];
update_col            := column_ref '=' expression;
delete_stmt           := DELETE FROM table_ref [where_clause];

create_expr_stmt      := CREATE NAMED EXPRESSION named_table_expr | named_action_expr;
drop_expr_stmt        := DROP NAMED EXPRESSION Identifier;

expression            := column_ref | value | sub_select | function_call | case_expr
                       | Placeholder
                       | '(' expression ')'
                       | expression Operation expression
                       | (NOT | '-' | '+') expression
                       | expression IS [NOT] NULL
                       | expression IN ( sub_select | value_list )
                       ;
```

```
case_expr             := CASE [expression] when_clause+ [else_clause] END;
when_clause           := WHEN expression THEN expression;
else_clause           := ELSE expression;

sub_select            := '(' select_stmt ')'
value_list            := '(' value (',' value)* ')';
action_stmt           := insert_stmt | update_stmt | delete_stmt;

attr_name             := Identifier;
table_name            := Identifier;
datastore             := Identifier;
col_alias             := Identifier;
table_alias           := Identifier;

schema_ref            := Identifier;
table_ref             := ( [schema_ref '.'] Identifier )
                       | ( table_name ['(' expression (',' expression)* ')'] );
action_ref            := action_name ['(' expression (',' expression)* ')'];
function_ref          := [schema_ref '.'] Identifier;
column_ref            := [table_ref '.'] Identifier ('.' Identifier)*;

function_call         := function_ref '(' [expression (',' expression)*] ')';
value                 := NULL | TRUE | FALSE | Value;
```