Design and Implementation of the CloudMdsQL Multistore System

Boyan Kolev[1], Carlyna Bondiombouy[1], Oleksandra Levchenko[1], Patrick Valduriez[1], Ricardo Jimenez[2], Raquel Pau[3] and Jose Pereira[4]

[1]*Inria and LIRMM, University of Montpellier, Montpellier, France*
[2]*LeanXcale and Universidad Politécnica de Madrid, Madrid, Spain*
[3]*Sparsity Technologies, Barcelona, Spain*
[4]*INESC TEC and University of Minho, Braga, Portugal*

Keywords: Cloud, Multistore System, Heterogeneous Data Stores, SQL and NoSQL Integration.

Abstract: The blooming of different cloud data management infrastructures has turned multistore systems to a major topic in the nowadays cloud landscape. In this paper, we give an overview of the design of a Cloud Multidatastore Query Language (CloudMdsQL), and the implementation of its query engine. CloudMdsQL is a functional SQL-like language, capable of querying multiple heterogeneous data stores (relational, NoSQL, HDFS) within a single query that can contain embedded invocations to each data store's native query interface. The major innovation is that a CloudMdsQL query can exploit the full power of local data stores, by simply allowing some local data store native queries (e.g. a breadth-first search query against a graph database) to be called as functions, and at the same time be optimized.

1 INTRODUCTION

The blooming of different cloud data management infrastructures, specialized for different kinds of data and tasks, has led to a wide diversification of DBMS interfaces and the loss of a common programming paradigm. This makes it very hard for a user to integrate and analyze her data sitting in different data stores, e.g. RDBMS, NoSQL, and HDFS. For example, a media planning application, which needs to find top influencers inside social media communities for a list of topics, has to search for communities by keywords from a key-value store, then analyze the impact of influencers for each community using complex graph database traversals, and finally retrieve the influencers' profiles from an RDBMS and an excerpt of their blog posts from a document database. The CoherentPaaS project (CoherentPaaS, 2013) addresses this problem, by providing a rich platform integrating different data management systems specialized for particular tasks, data and workloads. The platform is designed to provide a common programming model and language to query multiple data stores, which we herewith present.

The problem of accessing heterogeneous data sources has long been studied in the context of multidatabase and data integration systems (Özsu and Valduriez, 2011). More recently, with the advent of cloud databases and big data processing frameworks, the solution has evolved towards multistore systems that provide integrated access to a number of RDBMS, NoSQL and HDFS data stores through a common query engine. Data mediation SQL engines, such as Apache Drill, Spark SQL (Armbrust et al., 2015), and SQL++ provide common interfaces that allow different data sources to be plugged in (through the use of wrappers) and queried using SQL. The polystore BigDAWG (Duggan et al., 2015) goes one step further by enabling queries across "islands of information", where each island corresponds to a specific data model and its language and provides transparent access to a subset of the underlying data stores through the island's data model. Another family of multistore systems (DeWitt et al., 2013, LeFevre et al., 2014) has been introduced with the goal of tightly integrating big data analytics frameworks (e.g. Hadoop MapReduce) with traditional RDBMS, by sacrificing the extensibility with other data sources. However, since none of these approaches supports the ad-hoc usage of native queries, they do not preserve the full expressivity of an arbitrary data store's query language. But what we want to give the user is the ability to express powerful ad-hoc queries that exploit the full power of the different

Kolev, B., Bondiombouy, C., Levchenko, O., Valduriez, P., Jimenez-Peris, R., Pau, R. and Pereira, J.
Design and Implementation of the CloudMdsQL Multistore System.
In *Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016) - Volume 1*, pages 352-359
ISBN: 978-989-758-182-3

data store languages, e.g. directly express a path traversal in a graph database. Therefore, the current multistore solutions do not directly apply to solve our problem.

In this paper, we give an overview of the design of a Cloud multidatastore query language (CloudMdsQL) and the implementation of its query engine. CloudMdsQL is a functional SQL-like language, capable of querying multiple heterogeneous databases (e.g. relational, NoSQL and HDFS) within a single query containing nested subqueries (Kolev et al., 2015). Thus, the major innovation is that a CloudMdsQL query can exploit the full power of local data stores, by simply allowing some local data store native queries (e.g. a breadth-first search query against a graph database) to be called as functions, and at the same time be optimized based on a simple cost model, e.g. by pushing down select predicates, using bind join, performing join ordering, or planning intermediate data shipping. CloudMdsQL has been extended (Bondiombouy et al., 2015) to address distributed processing frameworks such as Apache Spark by enabling the ad-hoc usage of user defined map/filter/reduce operators as subqueries, yet allowing for pushing down predicates and bind join conditions.

2 LANGUAGE OVERVIEW

The CloudMdsQL language is SQL-based with the extended capabilities for embedding subqueries expressed in terms of each data store's native query interface (Kolev et al., 2015). 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, in order to handle non-flat and nested data, with basic operators over such composite datatypes.

Queries that integrate data from several data stores usually consist of subqueries and an integration `SELECT` statement. A subquery is defined as a named table expression, i.e. an expression that returns a table and has a name and signature. The signature defines the names and types of the columns of the returned relation. Thus, each query, although agnostic to the underlying data stores' schemas, is executed in the context of an ad-hoc schema, formed by all named table expressions within the query. A named table expression can be defined by means of either an SQL `SELECT` statement (that the query compiler is able to analyze and possibly rewrite) or a native expression (that the query engine considers as a black box and delegates its processing directly to the data store). For example, the following simple CloudMdsQL query contains two subqueries, defined by the named table expressions `T1` and `T2`, and addressed respectively against the data stores `rdb` (an SQL database) and `mongo` (a MongoDB database):

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

The purpose of this query is to perform relational algebra operations (expressed in the main `SELECT` statement) on two datasets retrieved from a relational and a document database. The two subqueries are sent independently for execution against their data stores in order the retrieved relations to be joined by the common query engine. The SQL table expression `T1` is defined by an SQL subquery, while `T2` is a native expression (identified by the special bracket symbols `{* *}`) expressed as a native MongoDB call. The subquery of expression `T1` is subject to rewriting by pushing into it the filter condition `y <= 3`, specified in the main `SELECT` statement, thus reducing the amount of the retrieved data by increasing the subquery selectivity. Note that subqueries to some NoSQL data stores can also be expressed as SQL statements; in such cases, the wrapper must provide the translation from relational operators to native calls.

CloudMdsQL allows named table expressions to be defined as Python functions, which is useful for querying data stores that have only API-based query interface. A Python expression yields tuples to its result set much like a user-defined table function. It can also use as input the result of other subqueries. Furthermore, named table expressions can be parameterized by declaring parameters in the expression's signature. For example, the following Python expression uses the intermediate data retrieved by `T2` to return another table containing the number of occurrences of the parameter `v` in the array `T2.z`.

```
T3(x int, c int
     WITHPARAMS v string)@python =
{*
  for (x, z) in CloudMdsQL.T2:
    yield( x, z.count(v) )
*}
```

A (parameterized) named table can then be instantiated by passing actual parameter values from another native/Python expression, as a table function in a `FROM` clause, or even as a scalar function (e.g. in the `SELECT` list). Calling a named table as a scalar function is useful e.g. to express direct lookups into a key-value data store.

Note that parametrization and nesting is also available in SQL and native named tables. For example, we validated the query engine with a use case that involves the Sparksee graph database and we use its Python API to express subqueries that benefit from all of the features described above (Kolev et al., 2015). In fact, our initial query engine implementation enables Python integration; however support for other languages (e.g. JavaScript) for user-defined operations can be easily added.

In order to also address distributed processing frameworks (such as Apache Spark) as data stores, we introduce a formal notation that enables the ad-hoc usage of user-defined Map/Filter/Reduce (MFR) operators as subqueries in CloudMdsQL to request data processing in an underlying big data processing framework (DPF) (Bondiombouy et al., 2015). 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)@hdfs = {*
   SCAN(TEXT,'words.txt')
  .MAP(KEY,1)
  .FILTER( KEY LIKE '%cloud%' )
  .REDUCE(SUM)
  .PROJECT(KEY,VALUE)
*}
```

This MFR subquery is also a subject to rewriting according to rules based on the algebraic properties of the MFR operators. In the example above, the `MAP` and `FILTER` operations will be swapped, thus allowing the filter to be applied earlier. The same rules apply for any pushed down predicates.

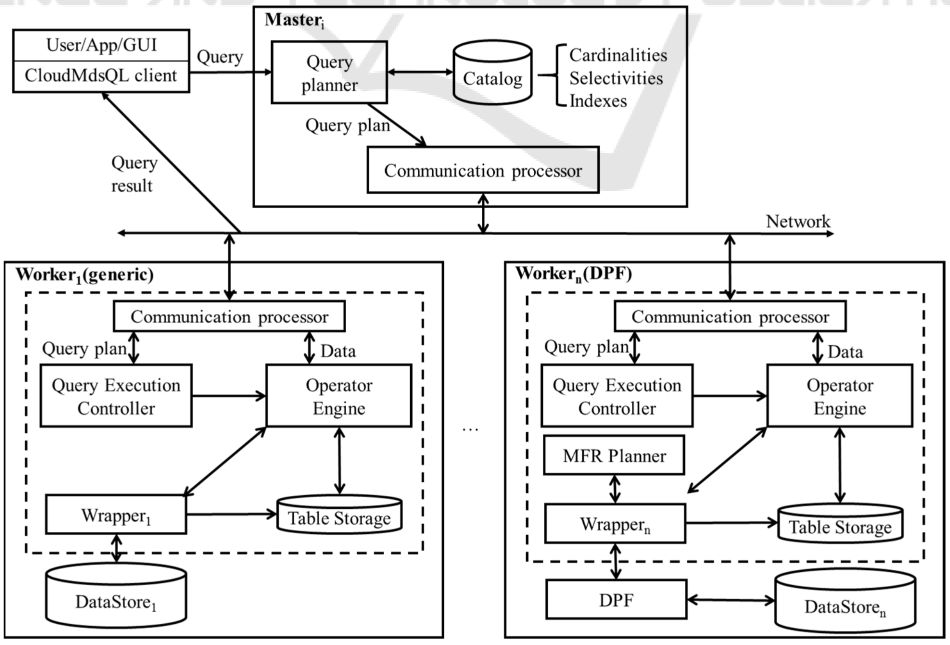


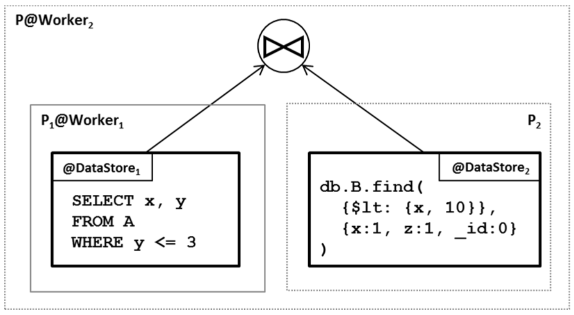
Figure 1: Architecture of the query engine.

3 SYSTEM OVERVIEW

In this section, we introduce the generic architecture of the query engine, with its main components. The design of the query engine takes advantage of the fact that it operates in a cloud platform, with full control over where the system components can be installed. The architecture of the query engine is fully distributed (see Figure 1), so that query engine nodes can directly communicate with each other, by exchanging code (query plans) and data.

Each query engine node consists of two parts – master and worker – and is collocated at each data store node in a computer cluster. Each master or worker has a communication processor that supports send and receive operators to exchange data and commands between nodes. To ease readability in Figure 1, we separate master and worker, which makes it clear that for a given query, there will be one master in charge of query planning and one or more workers in charge of query execution. To illustrate query processing with a simple example, let us consider a query Q on two data stores in a cluster with two nodes (e.g. the query introduced in Section 2). Then a possible scenario for processing Q, where the node id is written in subscript, is the following:

- At client, send Q to Master$_1$.
- At Master$_1$, produce a query plan P (see Figure 2) for Q and send it to Worker$_2$, which will control the execution of P.
- At Worker$_2$, send part of P, say P$_1$, to Worker$_1$, and start executing the other part of P, say P$_2$, by querying DataStore$_2$.
- At Worker$_1$, execute P$_1$ by querying DataStore$_1$, and send result to Worker$_2$.
- At Worker$_2$, complete the execution of P$_2$ (by integrating local data with data received from Worker$_1$), and send the final result to the client.



```
SELECT x, y
FROM A
WHERE y <= 3
```

```
db.B.find(
  {$lt: {x, 10}},
  {x:1, z:1, _id:0}
)
```

Figure 2: A simple query plan.

This simple example shows that query execution can be fully distributed among the two nodes and the result sent from where it is produced directly to the client, without the need for an intermediate node.

A **master** node takes as input a query and produces a query plan, which it sends to one chosen query engine node for execution. The query planner performs query analysis and optimization, and produces a query plan serialized in a JSON-based intermediate format that can be easily transferred across query engine nodes. The plan is abstracted as a tree of operations, each of which carries the identifier of the query engine node that is in charge of performing it. This allows us to reuse query decomposition and optimization techniques from distributed query processing (Özsu and Valduriez, 2011), which we adapt to our fully distributed architecture. In particular, we strive to:

- Minimize local execution time in the data stores, by pushing down select operations in the data store subqueries and exploiting bind join by subquery rewriting;
- Minimize communication cost and network traffic by reducing data transfers between workers.

To compare alternative rewritings of a query, the query planner uses a simple catalog, which is replicated at all nodes in primary copy mode. The catalog provides basic information about data store collections such as cardinalities, attribute selectivities and indexes, and a simple cost model. Such information can be given with the help of the data store administrators or collected by wrappers in local catalogs which are then merged into the global catalog. The query language provides a possibility for the user to define cost and selectivity functions whenever they cannot be derived from the catalog, mostly in the case of using native subqueries.

The query engine is designed to verify the executability of rewritten subqueries to data stores, e.g. due to selection pushdowns or usage of bind join. For this reason, each wrapper may provide the query planner with the capabilities of its data store to perform operations supported by the common data model.

Workers collaborate to execute a query plan, produced by a master, against the underlying data stores involved in the query. As illustrated above, there is a particular worker, selected by the query planner, which becomes in charge of controlling the execution of the query plan. This worker can subcontract parts of the query plan to other workers and integrate the intermediate results to produce the final result.

Each worker node acts as a lightweight runtime database processor atop a data store and is composed of three generic modules (i.e. same code library) - query execution controller, operator engine, and table storage - and one wrapper module that is specific to a data store. These modules provide the following capabilities:

- Query execution controller: initiates and controls the execution of a query plan (received from a master or worker) by interacting with the operator engine for local execution or with one or more workers (through communication processors).
- Operator engine: executes the query plan operators on data retrieved from the wrapper, from another worker, or from the table storage. The operator engine may write an intermediate relation to the table storage, e.g. when it needs to be consumed by more than one operator or when it participates in a blocking operation.
- Table Storage: provides efficient, uniform storage (main memory and disk) for intermediate and result data in the form of tables.
- Wrapper: interacts with its data store through its native API to retrieve data, transforms the result in the form of table, and writes the result in table storage or delivers it to the operator engine.

In addition to the generic modules, which are valid for all the workers, the components of a worker collocated with a data processing framework as data store, have some specifics. First, the worker implements a parallel operator engine, thus providing a tighter coupling between the query processor and the underlying DPF and hence taking more advantage of massive parallelism when processing HDFS data. Second, the wrapper of the distributed data processing framework has a slightly different behavior as it processes MFR expressions wrapped in native subqueries. It parses and interprets a subquery written in MFR notation; then uses an MFR planner to find optimization opportunities; and finally translates the resulting sequence of MFR operations to a sequence of DPF's API methods to be executed. The MFR planner decides where to position pushed down filter operations to apply them as early as possible, using rules for reordering MFR operators that take into account their algebraic properties.

4 IMPLEMENTATION

For the current implementation of the query engine, we modified the open source Derby database to accept CloudMdsQL queries and transform the corresponding execution plan into Derby SQL operations. We developed the query planner and the query execution controller and linked them to the Derby core, which we use as the operator engine. In this section, we focus on the implementation of the components, which each query engine node consists of (Figure 3).

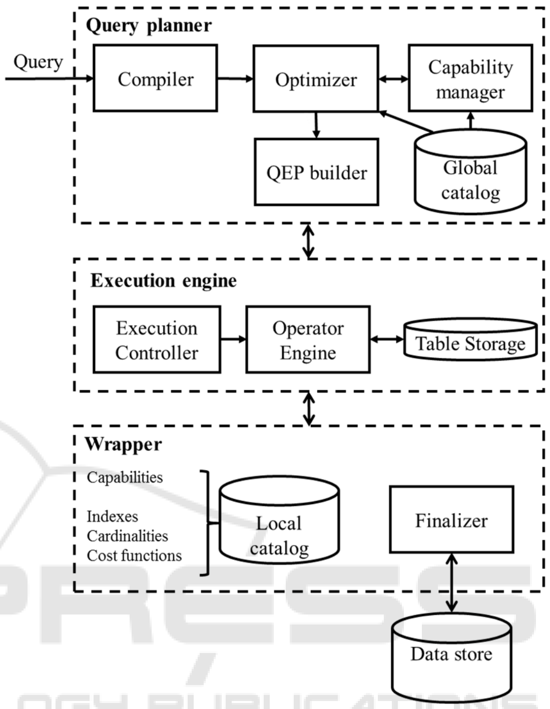


Figure 3: Query engine implementation components.

4.1 Query Planner

The query planner is implemented in C++; it compiles a CloudMdsQL query and generates a query execution plan (QEP) to be processed by the query execution engine. The result of the query planning is the JSON serialization of the generated QEP, which is represented as a directed acyclic graph, where leaf nodes are references to named tables and all other nodes represent relational algebra operations. The query planning process goes through several phases, which we briefly focus on below.

The query **compiler** uses the Boost.Spirit framework for parsing context-free grammars, following the recursive descent approach. Boost.Spirit allows grammar rules to be defined by means of C++ template metaprogramming techniques. Each grammar rule has an associated semantic action, which is a C++ function that should return an object, corresponding to the grammar rule.

The compiler first performs lexical and syntax analyses of a CloudMdsQL query to decompose it into an abstract syntax tree (AST) that corresponds to the syntax clauses of the query.

At this stage the compiler identifies a forest of sub-trees within the AST, each of which is associated to a certain data store (labelled by a named table) and meant to be delivered to the corresponding wrapper to translate it to a native query and execute it against the data store. The rest of the AST is the part that will be handled by the common query engine (the common query AST).

Furthermore, the compiler performs a semantic analysis of the AST by first resolving the names of tables and columns according to the ad-hoc schema of the query following named table signatures. Datatype analysis takes place to check for datatype compatibilities between operands in expressions and to infer the return datatype of each operation in the expression tree, which may be further verified against a named table signature, thus identifying implicit typecasts or type mismatches. WHERE clause analysis is performed to discover implicit equi-join conditions and opportunities for moving predicates earlier in the common plan. The cross-reference analysis aims at building a graph of dependencies across named tables. Thus the compiler identifies named tables that participate in more than one operation, which helps the execution controller to plan for storing such intermediate data in the table storage. In addition, the optimizer avoids pushing down operations in the sub-trees of such named tables. To make sure that the dependency graph has no cycles, hence the generated QEP will be a directed acyclic graph, the compiler implements a depth-first search algorithm to detect and reject any query that has circular references.

Error handling is performed at the compilation phase. Errors are reported as soon as they are identified (terminating the compilation execution), together with the type of the error and the context, within which they were found (e.g. unknown table or column, ambiguous column references, incompatible types in expression, etc.).

The query **optimizer** uses the cost information in the **global catalog** and implements a simple exhaustive search strategy to explore all possible rewritings of the initial query, by pushing down select operations, expressing bind joins, join ordering, and intermediate data shipping. Furthermore, it uses the **capability manager** to validate each rewritten subquery against its data store capability specification, which is exported by the wrapper into the global catalog in the form of a JSON schema. Thus, the capability manager simply serializes each sub-tree of the AST into a JSON object and attempts to validate it against the corresponding JSON schema. This allows a rewritten sub-plan to a data store to be validated by the query planner before it is actually delivered to the wrapper for execution; and in case the validation fails, the rewriting action (e.g. selection pushdown) is reverted.

The **QEP builder** is responsible for the generation of the final QEP, ready to be handled by the query execution engine, which also includes: resolving attribute names to column ordinal positions considering the named table expression signatures, removing columns from intermediate projections in case they are no longer used by the operations above (e.g. as a result of operation pushdown), and serializing the QEP to JSON.

4.2 Operator Engine

The main reasons to choose Derby database to implement the operator engine are because Derby:

- Allows extending the set of SQL operations by means of `CREATE FUNCTION` statements. This type of statements creates an alias, which an optional set of parameters, to invoke a specific Java component as part of an execution plan.
- Has all the relational algebra operations fully implemented and tested.
- Has a complete implementation of the JDBC API.
- Allows extending the set of SQL types by means of `CREATE TYPE` statements. It allows working with dictionaries and arrays.

Having a way to extend the available Derby SQL operations allows designing the resolution of the named table expressions. In fact, the query engine requires three different components to resolve the result sets retrieved from the named table expressions:

- `WrapperFunction`: To send the partial execution plan to a specific data store using the wrappers interfaces and retrieve the results.
- `PythonFunction`: To process intermediate result sets using Python code.
- `NestedFunction`: To process nested CloudMdsQL queries.

Named table expressions admit parameters using the keyword `WITHPARAMS`. However, the current implementation of the `CREATE FUNCTION` statement is designed to bind each parameter declared in the statement with a specific Java method parameter. In fact, it is not designed to work with Java methods that can be called with a variable number of parameters, which is a feature introduced since Java 6. To solve this gap, we have modified the internal validation of the `CREATE FUNCTION` statement and how to invoke Java methods with a variable number of parameters during the evaluation of the execution plan. For example, imagine that the user declares a named table expression `T1` that returns 2 columns (`x` and `y`) and has a parameter called `a` as follows:

```
T1(x int, y string
     WITHPARAMS a string)@db1 =
( SELECT x, y FROM tbl WHERE id = $a )
```

The query execution controller will produce dynamically the following `CREATE FUNCTION` statement:

```
CREATE FUNCTION T1 ( a VARCHAR( 50 ) )
RETURNS TABLE ( x INT, y VARCHAR( 50 ))
LANGUAGE JAVA
PARAMETER STYLE DERBY_JDBC_RESULT_SET
READS SQL DATA
EXTERNAL NAME 'WrapperFunction.execute'
```

It is linked to the following Java component, which will use the wrapper interfaces to establish a communication with the data store `db1`:

```
public class WrapperFunction {
     public static ResultSet execute(
            String namedExprName,
            Long queryId,
            Object... args
            /*dynamic args*/
      ) throws Exception {
      //Code to invoke the wrappers
      }
}
```

Therefore, after accepting the execution plan in JSON format, the query execution controller parses it, identifies the sub-plans within the plan that are associated to a named table expression and dynamically executes as many `CREATE FUNCTION` statements as named table expressions exist with a unique name. As a second step, the execution engine evaluates which named expressions are queried more than once and must be cached into the temporary table storage, which will be always queried and updated from the specified Java functions to reduce the query execution time. Finally, the last step consists of translating all operation nodes that appear in the execution plan into a Derby specific SQL execution plan. Once the SQL execution plan is valid, the Derby core (which acts as the operator engine) produces a dynamic byte code that resolves the query that can be executed as many times as the application needs.

Derby implements the JDBC interface and an application can send queries though the Statement class. So, when the user has processed the query result and closed the statement, the query execution controller drops the previously created functions and cleans the temporary table storage.

To process data in distributed data stores we used a specific implementation of the Operator Engine and the MFR wrapper, adapting the parallel SQL engine Spark SQL (Armbrust et al., 2015) to serve as the operator engine, thus taking full advantage of massive parallelism when joining HDFS with relational data. To do this, each execution (sub-)plan is translated to a flow of invocations of Spark SQL's DataFrame API methods.

4.3 Wrappers

The wrappers are Java classes implementing a common interface used by the operator engine to interact with them. A wrapper may store locally catalog information and capabilities, which it provides to the query planner periodically or on demand. Each wrapper also implements a finalizer, which translates a CloudMdsQL sub-plan to a native data store query.

We have validated the query engine using four data stores – Sparksee (a graph database with Python API), Derby (a relational database accessed through its Java Database Connectivity (JDBC) driver), MongoDB (a document database with a Java API), and unstructured data stored in an HDFS cluster and processed using Apache Spark as big data processing framework (DPF). To be able to embed subqueries against these data stores, we developed wrappers for each of them as follows.

The wrapper for Sparksee accepts as raw text the Python code that needs to be executed against the graph database using its Python client API in the environment of a Python interpreter embedded within the wrapper.

The wrapper for Derby executes SQL statements against the relational database using its JDBC driver. It exports an `explain()` function that the query planner invokes to get an estimation of the cost of a subquery. It can also be queried by the query planner about the existence of certain indexes on table columns and their types. The query planner may then cache this metadata information in the catalog.

The wrapper for MongoDB is implemented as a wrapper to an SQL compatible data store, i.e. it performs native MongoDB query invocations according to their SQL equivalent. The wrapper maintains the catalog information by running probing queries such as `db.collection.count()` to keep actual database statistics, e.g. cardinalities of document collections. Similarly to the Derby wrapper, it also provides information about available indexes on document attributes.

The MFR wrapper implements an MFR planner to optimize MFR expressions in accordance with any pushed down selections. The wrapper uses Spark's Python API, and thus translates each transformation to Python lambda functions. Besides, it also accepts raw Python lambda functions as transformation definitions. The wrapper executes the dynamically built Python code using the reflection capabilities of Python by means of the `eval()` function. Then, it transforms the resulting RDD into a Spark DataFrame.

5 CONCLUSIONS

In this paper, we presented CloudMdsQL, a common language for querying and integrating data from heterogeneous cloud data stores and the implementation of its query engine. By combining the expressivity of functional languages and the manipulability of declarative relational languages, it stands in "the golden mean" between the two major categories of query languages with respect to the problem of unifying a diverse set of data management systems. CloudMdsQL satisfies all the legacy requirements for a common query language, namely: support of nested queries across data stores, data-metadata transformations, schema independence, and optimizability. In addition, it allows embedded invocations to each data store's native query interface, in order to exploit the full power of data stores' query mechanism.

The architecture of CloudMdsQL query engine is fully distributed, so that query engine nodes can directly communicate with each other, by exchanging code (query plans) and data. Thus, the query engine does not follow the traditional mediator/wrapper architectural model where mediator and wrappers are centralized. This distributed architecture yields important optimization opportunities, e.g. minimizing data transfers by moving the smallest intermediate data for subsequent processing by one particular node. The wrappers are designed to be transparent, making the heterogeneity explicit in the query in favor of preserving the expressivity of local data stores' query languages. CloudMdsQL sticks to the relational data model, because of its intuitive data representation, wide acceptance and ability to integrate datasets by applying joins, unions and other relational algebra operations.

The CloudMdsQL query engine has been validated (Kolev et al., 2015; Bondiombouy et al., 2015) with four different database management systems – Sparksee (a graph database with Python API), Derby (a relational database accessed through its JDBC driver), MongoDB (a document database with a Java API) and Apache Spark (a parallel framework processing distributed data stored in HDFS, accessed by Apache Spark API). The performed experiments have evaluated the impact of the used optimization techniques on the overall query execution performance (Kolev et al., 2015; Bondiombouy et al., 2015).

REFERENCES

Armbrust, M., Xin, R., Lian, C., Huai, Y., Liu, D., Bradley, J., Meng, X., Kaftan, T., Franklin, M., Ghodsi, A., Zaharia, M. 2015. *Spark SQL: Relational Data Processing in Spark.* In ACM SIGMOD (2015), 1383-1394.

Bondiombouy, C., Kolev, B., Levchenko, O., Valduriez, P. 2015. *Integrating Big Data and Relational Data with a Functional SQL-like Query Language.* Int. Conf. on Databases and Expert Systems Applications (DEXA) (2015), 170-185.

CoherentPaaS, http://coherentpaas.eu (2013).

DeWitt, D., Halverson, A., Nehme, R., Shankar, S., Aguilar-Saborit J., Avanes, A., Flasza, M., Gramling, J. 2013. *Split Query Processing in Polybase.* In ACM SIGMOD (2013), 1255-1266.

Duggan, J., Elmore, A. J., Stonebraker, M., Balazinska, M., Howe, B., Kepner, J., Madden, S., Maier, D., Mattson, T., Zdonik, S. 2015. *The BigDAWG Polystore System.* SIGMOD Rec. 44, 2 (August 2015), 11-16.

Kolev, B., Valduriez, P., Bondiombouy, C., Jiménez-Peris, R., Pau, R., Pereira, J. 2015. CloudMdsQL: Querying Heterogeneous Cloud Data Stores with a Common Language. *Distributed and Parallel Databases*, pp 1-41, http://hal-lirmm.ccsd.cnrs.fr/lirmm-01184016.

LeFevre, J., Sankaranarayanan, J., Hacıgümüs, H., Tatemura, J., Polyzotis, N., Carey, M. 2014. *MISO: Souping Up Big Data Query Processing with a Multistore System.* In ACM SIGMOD (2014), 1591-1602.

Özsu, T., Valduriez, P. 2011. *Principles of Distributed Database Systems – Third Edition.* Springer, 850 pages.