The CloudMdsQL Multistore System

Boyan Kolev, Carlyna Bondiombouy, Patrick Valduriez, Ricardo Jiménez-Peris, Raquel Pau, José Pereira

▶ **To cite this version:**

Boyan Kolev, Carlyna Bondiombouy, Patrick Valduriez, Ricardo Jiménez-Peris, Raquel Pau, et al.. The CloudMdsQL Multistore System. ACM SIGMOD, Jun 2016, San Francisco, United States. 10.1145/2882903.2899400 . lirmm-01288025

HAL Id: lirmm-01288025

https://hal-lirmm.ccsd.cnrs.fr/lirmm-01288025

Submitted on 14 Mar 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

The CloudMdsQL Multistore System

Boyan Kolev, Carlyna Bondiombouy, Patrick Valduriez
Inria and LIRMM
Montpellier, France
firstname.lastname@inria.fr

Ricardo Jiménez-Peris
LeanXcale and UPM
Madrid, Spain

Raquel Pau
Sparsity Technologies
Barcelona, Spain

José Pereira
INESC TEC and U. Minho
Braga, Portugal

ABSTRACT
The blooming of different cloud data management infrastructures has turned multistore systems to a major topic in the nowadays cloud landscape. In this demonstration, we present a Cloud Multidatastore Query Language (CloudMdsQL), and its query engine. CloudMdsQL is a functional SQL-like language, capable of querying multiple heterogeneous data stores (relational and NoSQL) within a single query that may 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.

Within our demonstration, we focus on two use cases each involving four diverse data stores (graph, document, relational, and key-value) with its corresponding CloudMdsQL queries. The query execution flows are visualized by an embedded real-time monitoring subsystem. The users can also try out different ad-hoc queries, not necessarily in the context of the use cases.

Keywords
Cloud; multistore system; heterogeneous data stores; SQL and NoSQL integration.

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[1] 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 [7]. 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, 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 [3] 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 [2,6] 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 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 demonstration, we present Cloud multidatastore query language (CloudMdsQL), a functional SQL-like language, designed for querying multiple heterogeneous databases (e.g. relational and NoSQL) within a single query containing nested subqueries [5]. Each subquery addresses directly a particular data store and may contain embedded invocations to the data store's native query interface. 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3531-7/16/06…$15.00 .
DOI: http://dx.doi.org/10.1145/2882903.2899400

[1] http://coherentpaas.eu

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 [1] 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. 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, _id:0} )
*}
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. Note that subqueries to some NoSQL data stores can also be expressed as SQL statements; in such cases, the wrapper must provide the mapping from relational operators to native calls. In our demonstration, unlike in the example above, we use an SQL wrapper to query MongoDB, which also benefits from subquery rewriting.

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. In our demonstration, we give an example that involves the Sparksee graph database and we use its Python API to express subqueries that benefit from all of the features described above. 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.

3. SYSTEM OVERVIEW

The query engine follows a mediator/wrapper architecture. The query compiler decomposes the query into a query execution plan (QEP), which appears as a directed acyclic graph of relational operators where leaf nodes correspond to subqueries for the wrappers to execute directly against the data stores.

3.1 Query Optimization

Before its actual execution, a QEP may be rewritten by the query optimizer. To compare alternative rewritings of a query, the optimizer uses basic cost information exposed by the wrappers in the form of cost functions or database statistics, and a simple cost model. In addition, 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.

CloudMdsQL uses bind join as an efficient method for performing semi-joins across heterogeneous data stores that uses subquery rewriting to push the join conditions. For example, the list of distinct values of the join attribute(s), retrieved from the left-hand side subquery, is passed as a filter to the right-hand side subquery. To illustrate it, let us consider the following CloudMdsQL query:

```
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
```

Let us assume that the optimizer has decided to use the bind join method and that the join condition will be bound to the right-hand side of the join operation. First, the relation `B` is retrieved from the corresponding data store using its query mechanism. Then, the distinct values of `B.id` are used as a filter condition in the query that retrieves the relation `A` from its data store. Assuming that the distinct values of `B.id` are b_1 … b_n, the query to retrieve the right-hand side relation of the bind join uses the following SQL approach (or its equivalent according to the data store's query language), thus retrieving from `A` only the rows that match the join criteria:

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

The way to do the bind join analogue for native/Python queries is through the use of a `JOINED ON` clause in the named table signature. For example, if `A` is defined as the Python function below, as `A.id` participates in an equi-join, the values b_1 … b_n will be provided to the Python code through the iterator `Outer`:

```
A(id int, x int JOINED ON id)@DB1 = {*
  for id in CloudMdsQL.Outer:
    yield ( id, db.get_x(id) )
*}
```

3.2 Query Engine Implementation

For the current implementation of the query engine, we modified the open source Apache 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. Derby allows extending the set of SQL operations by means of `CREATE FUNCTION` statements. This type of statements creates an alias, with an optional set of parameters, to invoke a specific Java component as part of an execution plan. Thus, for each named table expression in a query, a table function is created dynamically, which invokes the corresponding wrapper as a Java class. Thus, Derby handles global execution, delegating local optimization and execution to the underlying data stores. As a second step, the query 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.

4. DEMONSTRATION

The demonstration concentrates on two CloudMdsQL use case scenarios from different information systems: a social network analysis tool for marketing companies and a bibliographic recommendation system. The users will have the possibility to experience the use case scenarios through their web interfaces. They will be also able to try out custom CloudMdsQL queries, to follow their corresponding query execution plans, and to monitor their execution flow through X-Ray [4] – a subsystem of the CoherentPaaS platform for real-time visualization of performance and resource usage integrated with all the components of the platform (the query engine and the underlying data stores).

Scenario 1. The first use case aims at finding the *communities in a social network, for a specific set of topics, with their top influencers*. Marketing companies are interested in discovering the people they need to convince about the quality of a specific brand. The dataset of this use case is a sample of Twitter, but it allows working with other social networks like Facebook or blogs. The application runs a Twitter listener of a set of topics in real-time; it modifies the database for each tweet it receives. The schema of this application contains a generic entity called *Document* to store text-items (tweets, messages, etc.), which can appear *copies* or *references*. An *Entity* (person or company) is an *author* of a document or a *mention* of a social-network account. The people interactions in social networks with copies, references or mentions, can be understood as a set of graph of influences. In other words, we can infer who influences who and about what. These *Influences* and the *Communities* are incrementally computed when a new tweet comes to the application and thus, these concepts are part of the application schema.

The specification of the main query Q_1 the application uses is as follows: given a set of keywords k_1, k_2, k_3, find the 10 biggest communities and, for each community, find the 20 most influencers. For each of these influencers, the system must return the number of influenced entities inside the community, the influencer's id, name and account creation date and the last published document.

In order to implement this use case, we use a graph database (Sparksee) to store the graph of *Influences* and compute the *Communities*; a relational database (MonetDB) for all the basic information about *Entities* and *Documents* (only metadata); a document database (MongoDB) to store the *Document* contents; and a key-value data store (HBase) to index communities per keyword. Following the execution plan for the CloudMdSQL query Q_1, the query engine first invokes an HBase query to retrieve the communities preliminarily computed for a specific keyword; then, for each community, runs a Sparksee query using the Sparksee Python API to find the top 20 influencers, the number of influenced entities inside the community, and the maximum influence propagation depth. Finally, the basic information of each influencer (id, name, account creation date) and the last published document is retrieved by running queries to MonetDB and MongoDB. Figure 1 summarizes the described execution plan using a notation where each box represents a table expression as a data store subquery with its signature and a fragment, (pseudo)statement, or description of the subquery.

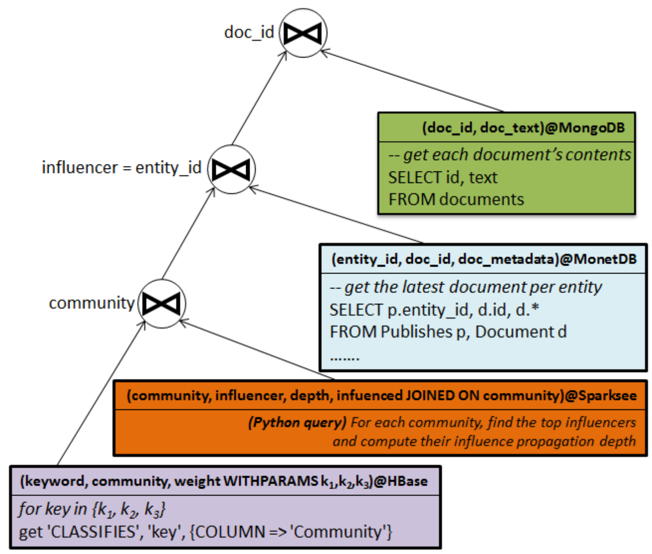


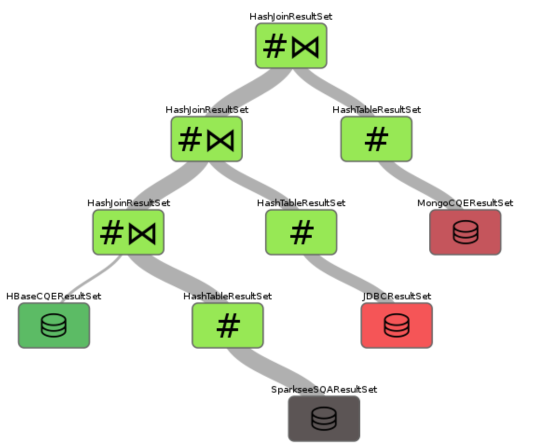
Figure 1. Execution plan for Q_1.

For this execution plan, the query optimization plays an important role to assign the bind join method to all the join operations. The reason is that the selected communities relevant to the keywords k_1, k_2 and k_3 are always a few, and thus the Sparksee query is evaluated only for a few communities, which significantly reduces the number of executions of expensive graph computations. Analogously, using bind join to retrieve the latest documents only for the filtered influencers increases the overall efficiency significantly by pushing bind join conditions to the MonetDB and MongoDB subqueries that take advantage of the existing indexes in both databases. Note that the MongoDB subquery is expressed in SQL, but the wrapper maps its sub-plan to a chain of invocations of MongoDB native API.

Within the results of this query, there is a nested level of information and the ranking of the suggested communities and influencers are important. For this reason, the Q_1 results are shown using a chart (see Figure 2) where the outer level of circles represents communities whereas the inner one corresponds to the influencers of those communities. The sizes of the community circles correspond to the relevance of the specified keywords with a community, while the sizes of the influencer circles correspond to the impact a person has on the community regarding the keywords.

Figure 2. Visualization of communities and influencers.

The query execution can be monitored using the integrated system for real-time monitoring and analysis X-Ray (see Figure 3), where the user can view details for each operation running within the process, including relative start/end times of operation executions, intermediate cardinalities, rewritten queries, etc.



		Name	Operands	Rows	Start	End	Comment
1	#⋈	HashJoinResultSet	[2,4]	223	0.026	0.043	
2	#	HashTableResultSet	[3]	196	0.026	0.041	key=[0]
3	⛁	MongoCQEResultSet	[]	196	0.023	0.025	SELECT id,text FRC
4	#⋈	HashJoinResultSet	[5,7]	223	0.025	0.040	
5	#	HashTableResultSet	[6]	196	0.025	0.040	key=[0]
6	⛁	JDBCResultSet	[]	196	0.020	0.023	SELECT p.entity_id,
7	#⋈	HashJoinResultSet	[8,10]	223	0.026	0.040	
8	#	HashTableResultSet	[9]	223	0.000	0.003	key=[0]
9	⛁	SparkseeSQAResultSet	[]	223	0.003	0.020	<python query>
10	⛁	HBaseCQEResultSet	[]	78	0.000	0.003	get 'CLASSIFIES', 'k

Figure 3. Monitoring of the query execution.

Scenario 2. The second use case application recommends *reviewers for a specific European project* taking into account the DBLP and CORDIS knowledge base. DBLP is a bibliographic dataset focused in computer science that currently contains 1,8 million publications and 1 million authors. CORDIS is the European projects dataset, which currently contains 40000 projects and 1000 institutions. The main query Q_2 is one of the key functionalities of a system built by Sparsity-Technologies to offer recommendations for researchers. The system visualizes the results from a web browser using HTML5 because it provides a clear way to analyze the results.

The schema of this information system contains *Projects*, whose *participants* are *Institutions* and one of them is the *coordinator*. On the other hand, a part of the schema stores a bibliographic dataset, which contains *Documents* (papers) and their *authors* (*People*) with the corresponding *affiliations* (*Institutions*) for each year. This information system also *indexes Projects* and *Documents* by *Keywords*; analyzes which are the top *expert Institutions* and *People* for each *Keyword*.

The application and the query Q_2 use a graph database (Sparksee) to resolve the conflicting interests with the members of the project because graph databases are efficient solving paths/joins; a relational data store (LeanXcale) to store and retrieve the complete list of fields about the recommended reviewers; a key-value data store (HBase) to find the top experts in a list of topics taking advantage of a fast search by keywords; and a document data store (MongoDB) to retrieve the contents of the last paper produced by the suggested reviewers.

The specification of Q_2 is as follows: given a specific project p and a set of keywords k_1, k_2, k_3, find the people that have never worked in the same institutions as the participants of p that are also experts in k_1, k_2, k_3. For these people, return their name, last affiliation and last paper title.

5. ACKNOWLEDGEMENTS

This research has been partially funded by the European Commission under projects CoherentPaaS and LeanBigData (grants FP7-611068, FP7-619606), the Madrid Regional Council, FSE and FEDER, project Cloud4BigData (grant S2013TIC-2894), and the Spanish Research Agency MICIN project BigDataPaaS (grant TIN2013-46883).

6. REFERENCES

[1] Bondiombouy, C., Kolev, B., Levchenko, O., Valduriez, P. 2015. Integrating Big Data and Relational Data with a Functional SQL-like Query Language. *DEXA*, 170-185.

[2] 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.

[3] 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. DOI=http://dx.doi.org/10.1145/2814710.2814713

[4] Guimarães, P., Pereira, J. 2015. X-Ray: Monitoring and Analysis of Distributed Database Queries, In *15th IFIP WG 6.1 International Conference, DAIS*, 80-93.

[5] 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.

[6] 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.

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