# Demonstration of the CloudMdsQL Multistore System

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

# Demonstration of the CloudMdsQL Multistore System

Boyan Kolev
Inria and LIRMM
Montpellier, France

Carlyna Bondiombouy
Inria and LIRMM
Montpellier, France

Patrick Valduriez
Inria and LIRMM
Montpellier, France

Ricardo Jiménez-Peris
LeanXcale and UPM
Madrid, Spain

Raquel Pau
Sparsity Technologies
Barcelona, Spain

José Pereira
INESC TEC and U. Minho
Braga, Portugal

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 has turned multistore systems to a major topic in the nowadays cloud landscape.

In this demonstration, we present Cloud multidatastore query language (CloudMdsQL) [1], a functional SQL-like language, designed for querying multiple heterogeneous databases (e.g. relational and NoSQL) within a single query containing nested subqueries. 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 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 or using bind join.

One of the major challenges in front of the CloudMdsQL language/engine is to allow joins across heterogeneous data stores and to be able to perform them in an efficient way. For this reason, we pay special attention to the use of bind joins and we apply this technique even when native queries are used.

This demonstration concentrates on a CloudMdsQL use case scenario: a social network analysis tool for marketing companies. The 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 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, 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.

For the 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.


*BDA* 2016, 15 au 18 Novembre, Poitiers, France.

## 1. REFERENCES

[1] B. Kolev, P. Valduriez, C. Bondiombouy, R. Jiménez-Peris, R. Pau, and J. Pereira. Cloudmdsql: querying heterogeneous cloud data stores with a common language. *Distributed and Parallel Databases*, 34(4):463–503, 2016.