Extending CloudMdsQL with MFR for Big Data Integration

Carlyna Bondiombouy, Boyan Kolev, Patrick Valduriez, Oleksandra Levchenko

Extending CloudMdsQL with MFR for Big Data Integration *

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

1. INTRODUCTION

Multistore systems [3] have been recently proposed to provide integrated access to multiple, heterogeneous data stores through a single query engine. Compared to multidatabase systems [6], multistore systems typically trade source autonomy for efficiency, using a tightly-coupled approach. In particular, much attention is being paid on the integration of unstructured big data (e.g. produced by web applications) typically stored in HDFS with relational data, e.g. in a data warehouse.

Existing solutions to integrate such unstructured and structured data do not directly apply to solve our problem, as they rely on having a relational view of the unstructured data, and hence require complex transformations. SQL engines, such as Hive, on top of distributed data processing frameworks are not always capable of querying unstructured HDFS data, thereby forcing the user to query the data by defining map/reduce functions.

Our approach is different as we propose a query language that can directly express subqueries that can take full advantage of the functionality of the underlying data processing frameworks. Furthermore, the language should allow for query optimization, so that the query operator execution sequence specified by the user may be reordered by taking into account the properties of map/filter/reduce operators together with the properties of relational operators. We respect the autonomy of the data stores and pay special attention to this throughout our experimental evaluation.

In this short paper (see [2] for the long version), we propose a functional SQL-like query language (based on CloudMdsQL) and query engine to retrieve data from two different kinds of data stores - an RDBMS and a distributed data processing framework such as Apache Spark or Hadoop MapReduce on top of HDFS - and combine them by applying data integration operators (mostly joins). However, users need to be aware of how data are organized across the data stores, so that they write valid queries. The query therefore contains embedded invocations to the underlying data stores, expressed as subqueries. As our query language is functional, it introduces a tight coupling between data and functions.

*Work partially funded by the European Commission under the Integrated Project CoherentPaaS [1].

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

2. QUERY LANGUAGE

The query language is based on a more general common query language, called CloudMdsQL [4], designed in the context of the CoherentPaaS project [1] to solve the problem of querying multiple heterogeneous databases (e.g. relational and NoSQL) within a single query while preserving the expressivity of their local query mechanisms. The common language itself 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 MongoDB arrays and JSON objects, in order to handle non-flat and nested data, with basic operators over such composite datatypes.

In this section, we introduce a formal notation to define Map/Filter/Reduce (MFR) subqueries in CloudMdsQL that request data processing in an underlying big data processing framework (DPF). Then we give an overview of how MFR statements are combined with SQL statements to express integration queries against a relational database and a DPF.

2.1 MFR Notation

An MFR statement represents a sequence of MFR operations on datasets. A dataset is considered simply as an abstraction for a set of tuples, where a tuple is a list of values, each of which can be a scalar value or another tuple. Although tuples can generally have any number of elements, mostly datasets that consist of key-value tuples are being processed by MFR operations. 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.

2.2 Combining SQL and MFR

Queries that integrate data from both a relational data store and a DPF usually consist of two subqueries (one expressed in SQL that addresses the relational database and another expressed in MFR that addresses the DPF) and an integration SELECT statement. The syntax follows the CloudMdsQL grammar introduced in [5]. 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 store' 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 compiler considers as a black box and passes to the wrapper as is, thus delegating it the processing of the subquery).

3. EXPERIMENTAL VALIDATION

The goal of our experimental validation is to evaluate the impact of query rewriting and optimization on execution time. More specifically, we explore the performance benefit of using bind join under different conditions.

3.1 Datasets

We generated data to populate the PostgreSQL table `scientists`, the MongoDB document collection `publications`, and text files with unstructured log data stored in HDFS.

3.2 Experimental Results

To evaluate the impact of optimization on query execution, we use a cluster of the GRID5000 platform (www.grid 5000.fr), with one node for PostgreSQL and MongoDB and 4 to 16 nodes for the HDFS cluster. The Spark cluster, used as both the DPF and the query processor, is collocated with the HDFS cluster. Each node in the cluster runs on 16 CPU cores at 2.4GHz, 64GB main memory, and the network bandwidth is 10Gbps. To demonstrate in detail the optimization techniques and their impact on the query execution, we prepared 1 query with different selectivity factors of the bind join condition. We execute the query in three different HDFS cluster setups - with 4, 8, and 16 nodes. Then we compare the execution times without and with bind join to the MFR subquery, which are illustrated in each query's corresponding graphical chart.

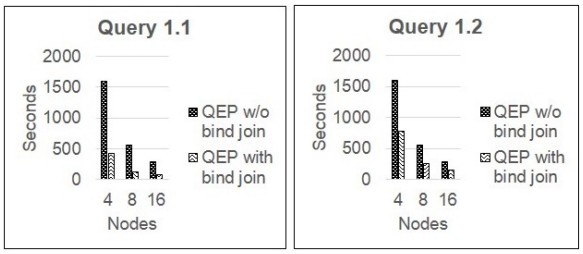
Query 1 involves all the data stores and aims at finding experts for publications of authors with a certain affiliation, it uses the MFR subquery `experts_alt` , which uses more sophisticated map functions, but makes only one shuffle, where the key is a keyword. For comparison, the MFR expression `experts` makes two shuffles, of which the first one uses a bigger key, composed of a keyword-author pair. Therefore, the corresponding Spark computation of Query 1 involves much smaller size of data to be shuffled, which explains its better overall efficiency and higher relative benefit of using bind join.

–Query 1.1: selectivity factor 10%

```
SELECT p.author, p.title, e.kw, e.expert
FROM scientists s, publications p, experts_alt e
WHERE s.affiliation = 'affiliation1'
AND p.author = s.name AND e.kw IN p.keywords
```

–Query 1.2: selectivity factor 30%

```
SELECT p.author, p.title, e.kw, e.expert
FROM scientists s, publications p, experts_alt e
WHERE s.affiliation IN ('affiliation1',
  'affiliation2', 'affiliation3')
AND p.author = s.name AND e.kw IN p.keywords
```



This experimental evaluation illustrates the query engine's ability to perform optimization and choose the most efficient execution plan. The results show the significant benefit of performing bind join in our experimental scenario, despite the overhead it produces (see [2]).

4. CONCLUSION

In this short paper, we proposed a functional SQL-like query language and query engine to integrate data from relational, NoSQL, and big data stores (such as HDFS). Our validation demonstrates that the proposed query language achieves the following requirements. First, it provides high expressivity by allowing the ad-hoc usage of specific map/filter/reduce operators through the MFR notation, as it was demonstrated with the `hdfs` subqueries. Second, it is optimizable as was demonstrated through performing bind join by rewriting the MFR subquery after retrieving the dataset from the MongoDB database. Our performance evaluation illustrates the query engine's ability to optimize a query and choose the most efficient execution strategy.

5. REFERENCES

[1] Coherentpaas project. `coherentpaas.eu`.

[2] C. Bondiombouy, B. Kolev, O. Levchenko, and P. Valduriez. Multistore big data integration with cloudmdsql. *TLDKS*, 28:48–74, 2016.

[3] C. Bondiombouy and P. Valduriez. Query processing in multistore systems: an overview. *International Journal of Cloud Computing (IJCC)*, 38, 2016.

[4] B. Kolev, C. Bondiombouy, P. Valduriez, R. Jiménez-Peris, R. Pau, and J. Pereira. The cloudmdsql multistore system. In *ACM SIGMOD Int. Conf. on Data Management*, pages 2113–2116, 2016.

[5] 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.

[6] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.