Multistore Big Data Integration with CloudMdsQL

Carlyna Bondiombouy, Boyan Kolev, Oleksandra Levchenko, Patrick Valduriez

Multistore Big Data Integration with CloudMdsQL

Carlyna Bondiombouy, Boyan Kolev, Oleksandra Levchenko, Patrick Valduriez

Inria and LIRMM, University of Montpellier, France
firstname.lastname@inria.fr

Abstract. Multistore systems have been recently proposed to provide integrated access to multiple, heterogeneous data stores through a single query engine. In particular, much attention is being paid on the integration of unstructured big data typically stored in HDFS with relational data. One main solution is to use a relational query engine that allows SQL-like queries to retrieve data from HDFS, which requires the system to provide a relational view of the unstructured data and hence is not always feasible. In this paper, we propose a functional SQL-like query language (based on CloudMdsQL) that can integrate data retrieved from different data stores, to take full advantage of the functionality of the underlying data processing frameworks by allowing the ad-hoc usage of user defined map/filter/reduce operators in combination with traditional SQL statements. Furthermore, our solution allows for optimization by enabling subquery rewriting so that bind join can be used and filter conditions can be pushed down and applied by the data processing framework as early as possible. We validate our approach through implementation and experimental validation with three data stores and representative queries. The experimental results demonstrate the usability of the query language and the benefits from query optimization.

1 Introduction

A major trend in cloud computing and big data is the understanding that there is "no one size fits all" solution. Thus, there has been a blooming of different cloud data management solutions, such as NoSQL, distributed file systems (e.g. Hadoop HDFS), and big data processing frameworks (e.g. Hadoop MapReduce or Apache Spark), specialized for different kinds of data and able to perform orders of magnitude better than traditional RDBMS. However, this has led to a wide diversification of data store 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. To address this problem, multistore systems [1, 8, 9, 11, 12, 13, 14, 15] have been recently proposed to provide integrated access to multiple, heterogeneous data stores through a single query engine.

Compared to multidatabase systems [16], 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.

One main solution is to use a relational query engine (e.g. Apache Hive) on top of a data processing framework (e.g. Hadoop MapReduce), which allows SQL-like queries to retrieve data from HDFS. However, this requires the system to provide a relational view of the unstructured data, which is not always feasible. In case the data store is managed independently from the relational query processing system, complex data transformations may need to take place (e.g. by applying specific map-reduce jobs) before the data can be processed by means of relational operators. Let us illustrate the problem, which will be the focus of this paper, with the following scenario.

Example scenario. An editorial office needs to find appropriate reporters for a list of publications based on given keywords. For the purpose, the editors need an analysis of the logs from a scientific forum stored in a Hadoop cluster in the cloud to find experts in a certain research field, considering the users who have mentioned particular keywords most frequently; and these results must be joined to the relational data in an RDBMS containing author and publication information. However, the forum application keeps log data about its posts in a non-tabular structure (the left side of the example below), namely in text files where a single record corresponds to one post and contains a fixed number of fields about the post itself (timestamp and username in the example) followed by a variable number of fields storing the keywords mentioned in the post.

```
2014-12-13, alice, storage, cloud
2014-12-22, bob, cloud, virtual, app      ⟶
2014-12-24, alice, cloud
```

KW	expert	freq
cloud	alice	2
storage	alice	1
virtual	bob	1
app	bob	1

The unstructured log data needs to be transformed into a tabular dataset containing for each keyword the expert who mentioned it most frequently (the right side of the example above). 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. Then the result dataset will be ready to be joined with the publication data retrieved from the RDBMS in order to suggest an appropriate reviewer for each publication. Being able to request such data processing with a single query is the scenario that motivates our work. However, the challenge in front of the query processor is optimization, i.e. it should be able of analyzing the operator execution flow of a query and performing operation reordering to take advantage of well-known optimization techniques (e.g. selection pushdowns and use of semi-joins) in order to yield efficient query execution.

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 optimiza-

tion, 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. This is especially useful for applying efficient query optimization by exploiting bind joins [10]; and we pay special attention to this throughout our experimental evaluation. Finally, we want to respect the autonomy of the data stores, e.g. HDFS and RDBMS, so that they can be accessible and controlled from outside our query engine with their own interface.

In this paper, 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). We assume that each data store is fully autonomous, i.e. the query engine has no control over the structure and organization of data in the data stores. For this reason, the architecture of our query engine is based on the traditional mediator/wrapper architectural approach [21] that abstracts the query engine from the specifics of each of the underlying data stores. However, users need to be aware of how data are organized across the data stores, so that they write valid queries. A single query of our language can request data to be retrieved from both stores and then a join to be performed over the retrieved datasets. 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. A subquery, addressing the data processing framework, is represented by a sequence of map/filter/reduce operations, expressed in a formal notation. On the other hand, SQL is used to express subqueries that address the relational data store as well as the main statement that performs the integration of data retrieved by all subqueries. Thus, a query benefits from both high expressivity (by allowing the ad-hoc usage of user defined map/filter/reduce operators in combination with traditional SQL statements) and optimizability (by enabling subquery rewriting so that bind join and filter conditions can be pushed inside and executed at the data store as early as possible).

This paper is a major extension of [4], with an improved generic architecture of the query engine (to support a wider range of underlying data models and to provide a tighter coupling with the data processing framework), a real experimental validation with three data stores (relational, document, and HDFS) and queries across them, and a more detailed comparison with the state of the art.

The rest of this paper is organized as follows. Section 2 introduces the language and its notation to express map/filter/reduce subqueries. Section 3 presents the architecture of the query engine. Section 4 elaborates more on the query processing and presents the properties of map/filter/reduce operators that constitute rewrite rules to perform query optimization. Section 5 gives a use case example walkthrough. Section 6 presents an experimental validation with (semi-)structured data stored in PostgreSQL and MongoDB, and unstructured data stored in an HDFS cluster and processed using Apache Spark. Section 7 discusses related work. Section 8 concludes.

2 Query Language

The query language is based on a more general common query language, called CloudMdsQL [12], designed in the context of the CoherentPaaS project [7] 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. Notice that the data processing defined in an MFR statement is not executed by the query engine, but is meant to be translated to a sequence of invocations to API functions of the DPF. In this paper, we use Apache Spark as an example of DPF, but the concept can be generalized to a wider range of frameworks that support the MapReduce programming model (such as Hadoop MapReduce, CouchDB, etc.).

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.

Core operators. The MAP operator produces key-value tuples by performing a specified transformation on the input tuples. The transformation is defined as an SQL-like expression that will be evaluated for each tuple of the input data set and should return a pair of values. The special variable `TUPLE` refers to the input tuple and its elements are addressed using a bracket notation. Moreover, the variables `KEY` and `VALUE` may be used as aliases to `TUPLE[0]` and `TUPLE[1]` respectively. The FILTER operator

selects from the input tuples only those, for which a specified condition is evaluated to *true*. The filter condition is defined as a boolean expression using the same special variables `TUPLE`, `KEY`, and `VALUE`. The REDUCE operator performs aggregation on values associated with the same key and produces a key-value dataset where each key is unique. The reduce transformation may be specified as an aggregate function (`SUM`, `AVG`, `MIN`, `MAX` or `COUNT`). Similarly to MAP, two other mapping operators are introduced: FLAT_MAP may produce numerous output tuples for a single input tuple; and MAP_VALUES defines a transformation that preserves the keys, i.e. applicable only to the values.

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 list of words. To count the words that contain the string 'cloud', we write the following composition of MFR operations:

```
MAP(KEY, 1).FILTER( KEY LIKE '%cloud%' ).REDUCE( SUM )
```

The first operation transforms each tuple (which has a single word as its only element) of the input dataset into a key-value pair where the word is mapped to a value of 1. The second operation selects only those key-value pairs for which the key contains the string 'cloud'. And the third one groups all tuples by key and performs a sum aggregate on the values for each key.

To process this statement, the query engine first looks for opportunities to optimize the execution by operator reordering. By applying MFR rewrite rules (explained in detail in Section 4.2), it finds out that the FILTER and MAP operations may be swapped so that the filtering is applied at an earlier stage. Further, it translates the sequence of operations into invocations of the underlying DPF's API. Notice that whenever a REDUCE transformation function has the associative property (like the SUM function), an additional combiner function call may be generated that precedes the actual reducer, so that as much data as possible will be reduced locally; e.g., this would be valid in the case of Hadoop MapReduce as the DPF, because it does not automatically perform local reduce. In the case of Apache Spark as the DPF, the query engine generates the following Python fragment to be included in a script that will be executed in Spark's Python environment:

```
dataset.filter( lambda k: 'cloud' in k ) \
       .map( lambda k: (k, 1) ) \
       .reduceByKey( lambda a, b: a + b )
```

In this example, all the MFR operations are translated to their corresponding Spark functions and all transformation expressions are translated to Python anonymous functions. In fact, to increase its expressivity, the MFR notation allows direct usage of anonymous functions to specify transformation expressions. This allows user-defined mapping functions, filter predicates, or aggregates to be used in an MFR statement. The user, however, needs to be aware of how the query engine is configured to interface the DPF, in order to know which language to use for the definition of inline anonymous functions (e.g. Spark may be used with Python or Scala, CouchDB – with JavaScript, etc.).

Input/output operators are normally used for transformation of data before and after the core map/filter/reduce execution chain. The SCAN operator loads data from its storage and transforms it to a dataset ready to be consumed by a core MFR operator. The PROJECT operator converts a key-value dataset to a tabular dataset ready to be involved in relational operations.

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 [12]. 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 compiler considers as a black box and passes to the wrapper as is, thus delegating it the processing of the subquery).

In this paper, we extend the usability of CloudMdsQL by adding the capability of handling MFR subqueries against DPFs and combining them with subqueries against other data stores. This is done in full compliance with CloudMdsQL properties, such as the ability to express nested subqueries (so that the output of one subquery, e.g. against an RDBMS, can be used as input to another subquery, e.g. MFR) which we further illustrate by the usage of bind joins. MFR subqueries are expressed as native named table expressions; this means that they are passed to their corresponding wrappers to process them (explained in more detail in Section 3).

In general, a single query can address a number of data stores by containing several named table expressions. We will now illustrate with a simple example how SQL and MFR statements can be combined, and in Section 5 will focus on a more sophisticated example involving 3 data stores. The following sample query contains two subqueries, defined by the named table expressions T1 and T2, and addressed respectively against the data stores aliased with identifiers `rdb` (for the SQL database) and `hdfs` (for the DPF):

```
T1(title string, kw string)@rdb = ( SELECT title, kw FROM tbl )
T2(word string, count int)@hdfs = {*
   SCAN(TEXT,'words.txt')
        .MAP(KEY,1).REDUCE(SUM).PROJECT(KEY,VALUE)
*}
SELECT title, kw, count FROM T1 JOIN T2 ON T1.kw = T2.word
WHERE T1.kw LIKE '%cloud%'
```

The purpose of this query is to perform relational algebra operations (expressed in the main SELECT statement) on two datasets retrieved from a relational database and a DPF. The two subqueries are sent independently for execution against their data

stores in order the retrieved relations to be joined by the query engine. The SQL table expression `T1` is defined by an SQL subquery. `T2` is an MFR expression that requests data retrieval from a text source and data processing by the specified map/reduce operations. Both subqueries are subject to rewriting by pushing into it the filter condition `kw LIKE '%cloud%'`, specified in the main SELECT statement, thus reducing the amount of the retrieved data by increasing the subquery selectivity and the overall efficiency. The so retrieved datasets are then converted to relations following their corresponding signatures, so that the main SELECT statement can be processed with semantic correctness. The PROJECT operator in the MFR statement provides a mapping between the dataset fields and the named table expression columns.

3 Generic Query Engine Architecture

The dominant state-of-the-art architectural model that addresses the problem of data integration and query processing across a diverse set of data stores is the mediator/wrapper architecture. A mediator is a software module that exploits encoded knowledge about certain sets or subsets of data to create information for a higher layer of applications [16]. In addition, a wrapper or adapter is a software component that encapsulates and hides the underlying complexity of sets or subsets of data by means of well-defined interfaces (it establishes communication and a data flow between mediators and data stores). In this section, we briefly describe the generic architecture of our system with an overview of the required steps to process a query.

The query language presented hereby assumes a query engine that follows the traditional mediator/wrapper architectural approach. By explicitly naming a data store identifier in a named table expression's signature, the query addresses the specific wrapper that is preliminarily configured and responsible for handling subqueries against the corresponding data store. Thus, a query can express an integration of data across several data stores, and in particular, integration of structured (relational DB), semi-structured (document DB), and unstructured (distributed storage, based on HDFS) data, which is the case that we focus on throughout our experimental validation.

Fig. 1 depicts the corresponding system architecture, containing a CloudMdsQL compiler, a common query processor (the mediator), three wrappers, and the three data stores – a distributed data processing framework (DPF), an RDBMS, and a document data store. The DPF is in charge of performing parallel data processing over a distributed data store. In this architecture, each data source has an associated wrapper that is responsible for executing subqueries against the data store and converting the retrieved datasets to tables matching the requested number and types of columns, so that they are ready to be consumed by relational operators at the query processor. The query processor consumes the query execution plan generated by the compiler and interacts with the wrappers through a common interface to: request handling of subqueries, centralize the information provided by the wrappers, and integrate the subqueries' results. The wrappers transform subqueries provided via the common interface into queries for the data stores. This generic architecture gives us the possi-

bility to use a specific implementation of the query processor and DPF wrapper, while reusing the CloudMdsQL query compiler and wrappers for relational and document data stores [12]. Although we can also reuse the CloudMdsQL query engine that has a distributed architecture [12], in our experimental work we explore the possibility to adapt the parallel SQL engine Spark SQL [2] to serve as the query processor, thus providing a tighter coupling between the query processor and the underlying DPF and hence taking more advantage of massive parallelism when joining HDFS with relational and document data.

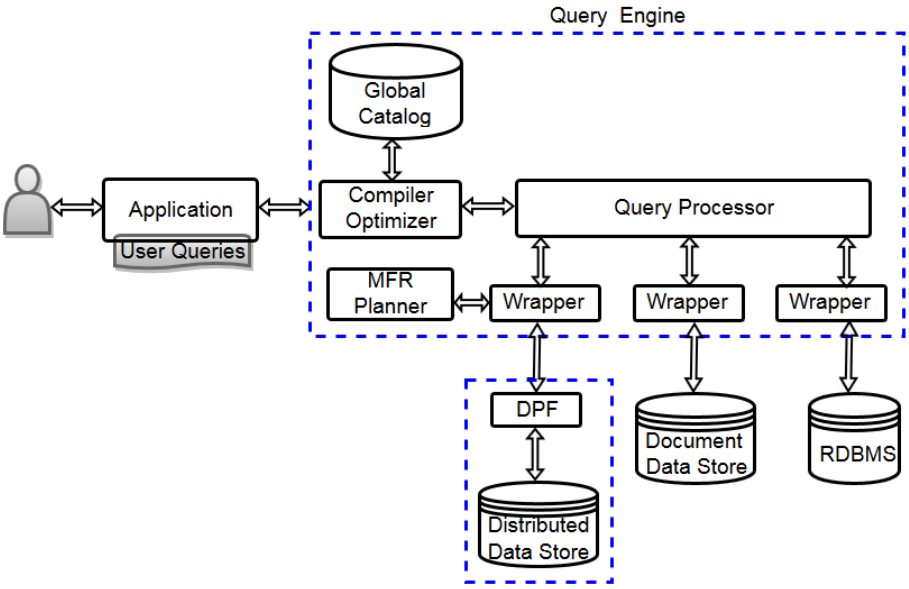


Fig. 1. Basic architecture of the query engine

Each of the wrappers is responsible for completing the execution of subqueries and retrieving the results. Upon initialization, each wrapper may provide to the query compiler the capability of its data store to process pushed down operations [12]. In our setup, all the three wrappers can accept pushdowns of filter predicates. Both the relational and document data store wrappers accept requests from the query processor in the form of query execution sub-plans represented as trees of relational algebra operators, resulting from the compilation of the SELECT statements expressed in the corresponding SQL named table expressions. The sub-plans may include selection operations resulting from pushed down predicates. The wrapper of the relational database has to build a SELECT statement out of a query sub-plan and to run it against its data store; then it retrieves the datasets and delivers them to the query processor in the corresponding format. The wrapper of the document data store (in our case, MongoDB) has to translate the sequence of relational operators from a query sub-plan to the corresponding sequence of MongoDB API calls; then it converts the resulting documents to tuples that match the signature of the corresponding named table expression. [12]

The wrapper of the distributed data processing framework has a slightly different behavior as it processes MFR expressions wrapped in native subqueries. First it parses and interprets a subquery written in MFR notation; then uses the 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. Once a dataset is retrieved as a result of the subquery execution, the wrapper provides it to the query processor in the format requested by the corresponding named table expression signature. The MFR planner decides where to position pushed down operations; e.g. it applies rules for MFR operator reordering to find the optimal place of a filter operation in order to apply it as early as possible and thus to reduce the query execution cost. To search for alternative operation orderings, the planner takes into account MFR rewrite rules, introduced in next section.

4 Query Processing

The query compiler first decomposes the query into a preliminary query execution plan (QEP), which, in its simplest form, is a tree structure representing relational operations. At this step, the compiler also identifies sub-trees within the query plan, each of which is associated to a certain data store. Each of these sub-plans is meant to be delivered to the corresponding wrapper, which has to translate it to a native query and execute it against its data. The rest of the QEP is the common plan that will be handled by the query engine.

4.1 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 a simple catalog, which provides basic information about data store collections such as cardinalities, attribute selectivities and indexes, and a simple cost model. Because of the autonomy of the underlying data stores, in order to derive local cost models, various classical black-box approaches for heterogeneous cost modeling, such as probing [26] and sampling [25, 27], have been adopted by the query optimizer. Thus, cost information can be collected by the wrappers and exposed to the optimizer in the form of cost functions or database statistics. Furthermore, the query language allows for user-defined cost and selectivity functions. And in case of lack of any cost information, heuristic rules are applied.

In our concrete example scenario with PostgreSQL, MongoDB, and MFR subqueries, we use the following strategy. The query optimizer executes an EXPLAIN request to PostgreSQL to directly estimate the cost of a subquery. The MongoDB wrapper runs in background probing queries to collect cardinalities of document collections, index availabilities, and index value distributions (to compute selectivities) and caches them in the query engine's catalog. As for an MFR subquery, if there is no user-provided cost information, the optimizer assumes that it is more

expensive than SQL subqueries and plans it at the end of the join order, which would also potentially benefit from the execution of bind joins.

The search space explored for optimization is the set of all possible rewritings of the initial query, by pushing down select operations, expressing bind joins, and join ordering. Unlike in traditional query optimization where many different permutations are possible, this search space is not very large, so we use a simple exhaustive search strategy.

Subquery rewriting takes place in order to request early execution of some operators and thus to increase its overall efficiency. Although several operations are subject to pushdowns across subqueries, in this paper we concentrate on the inclusion of only filter operations inside an MFR subquery. Generally, this is done in two stages: first, the query processor determines which operations can be pushed down for remote execution at the data stores; and second, the MFR planner may further determine the optimal place for inclusion of pushed down operations within the MFR operator chain by applying MFR rewrite rules (explained later in this section). Pushing a selection operation inside a subquery, either in SQL query or MFR operation chain, is usually considered beneficial, because it delegates the selection directly to the data store, which allows for early reducing of the size of data processed and retrieved from the data stores.

4.2 MFR Rewrite Rules

In this section, we introduce and enumerate some rules for reordering of MFR operators, based on their algebraic properties. These rules are used by the MFR planner to optimize an MFR subquery after a selection pushdown takes place.

Rule #1 (name substitution): upon pushdown, the filter is included just before the `PROJECT` operator and the filter predicate expression is rewritten by substituting column names with references to dataset fields as per the mapping defined by the `PROJECT` expressions. After this initial inclusion, other rules apply to determine whether it can be moved even farther. Example:

```
T1(a int, b int)@db1 ={* … .PROJECT(KEY, VALUE[0]) *}
SELECT a, b FROM T1 WHERE a > b
```

is rewritten to:

```
T1(a int, b int)@db1 ={* … .FILTER(KEY>VALUE[0]).PROJECT(KEY,VALUE[0])*}
SELECT a, b FROM T1
```

Rule #2: `REDUCE(<transformation>).FILTER(<predicate>)` is equivalent to `FILTER(<predicate>).REDUCE(<transformation>)`, if `predicate` condition is a function only of the `KEY`, because thus, applying the `FILTER` before the `REDUCE` will preserve the values associated to those keys that satisfy the filter condition as they would be if the `FILTER` was applied after the `REDUCE`. Analogously, under the same conditions, `MAP_VALUES(<transformation>).FILTER(<predicate>)` is equivalent to `FILTER(<predicate>).MAP_VALUES(<transformation>)`.

Rule #3: `MAP(<expr_list>).FILTER(<predicate1>)` is equivalent to `FILTER(<predicate2>).MAP(<expr_list>)`, where `predicate1` is rewritten to `predicate2` by substituting `KEY` and `VALUE` as per the mapping defined in `expr_list`. Example:

```
MAP(VALUE[0], KEY).FILTER(KEY > VALUE) →
FILTER(VALUE[0] > KEY).MAP(VALUE[0], KEY)
```

Since planning a filter as early as possible always increases the efficiency, the planner always takes advantage of moving a filter by applying rules #2 and #3 whenever they are applicable.

4.3 Bind Join

Bind join [10] is an efficient method for implementing semi-joins across heterogeneous data stores that uses subquery rewriting to push the join conditions. In this paper, we adapt the bind join approach for MFR subqueries and we focus on it in our experimental evaluation, as it brings a significant performance gain in certain occasions.

Using bind join between relational data (expressed in an SQL named table expression) and big data (expressed in an MFR named table expression) allows for reducing the computation cost at the DPF and the communication cost between the DPF and the query engine. This approach implicates that the list of distinct values of the join attribute(s) from the relation, preliminarily retrieved from the relational data store, is passed as a filter to the MFR subquery. To illustrate the approach, let us consider the following SELECT statement performing a join between an SQL named table `R` and an MFR named table `H`:

```
SELECT H.x, R.y FROM R JOIN H ON R.id = H.id WHERE R.z='abc'
```

To process this query using the bind join method, first, the table `R` is retrieved from the relational data store; then, assuming that the distinct values of `R.id` are $r_1 \ldots r_n$, the condition `id IN` $(r_1,\ldots,r_n)$ is passed as a `FILTER` to the MFR subquery that retrieves the dataset `H` from HDFS data store. Thus, only the tuples from `H` that match the join criteria are retrieved. Moreover, if the filter condition can be pushed even further in the MFR chain (according to the MFR rewrite rules) and thus to overcome at least one `REDUCE` operation, this may lead to a significant performance boost, as data will be filtered before at least one shuffle phase.

To estimate the expected performance gain of a bind join, the query optimizer takes into account the overhead a bind join may produce. First, when using bind join, the query engine must wait for the SQL named table to be fully retrieved before initiating the execution of the MFR subquery. Second, if the number of distinct values of the join attribute is large, using a bind join may slower the performance as it requires data to be pushed into the MFR subquery. In the example above, the query engine first asks the RDBMS (e.g. by running an `EXPLAIN` statement) for an estimation of the cardinality of data retrieved from `R`, after rewriting the SQL subquery by including the selection condition `R.z='abc'`. If the estimated cardinality does not exceed a

certain threshold, the optimizer plans for performing a bind join that can significantly increase the MFR subquery selectivity and affect the volume of transferred data.

5 Use Case Example

In this section, we reveal the steps the query engine takes to process a query using selection pushdown and especially bind join as optimization techniques. We also focus on the way the query engine dynamically rewrites the MFR subquery to perform a bind join. We consider three distinct data stores: PostgreSQL as the relational database (referred to as `rdb`), MongoDB as the document database (referred to as `mongo`) which will be subqueried by SQL expressions that are mapped by the wrapper to MongoDB calls, and an HDFS cluster (referred to as `hdfs`) processed using the Apache Spark framework.

Datasets. For the use case walkthrough we consider small sample datasets in the context of the multistore query example described in Section 1.

The `rdb` database stores structured data about scientists and their affiliations in the following table:

`Scientists:`

Name	**Affiliation**	**Country**
Ricardo	UPM	Spain
Martin	CWI	Netherlands
Patrick	INRIA	France
Boyan	INRIA	France
Larri	UPC	Spain
Rui	INESC	Portugal

The `mongo` database contains a document collection about publications including their keywords as follows:

```
Publications(
{ title:'Snapshot Isolation in Cloud DBs',     author:'Ricardo',
                keywords: ['transaction', 'cloud'] },
{ title:'Principles of Distributed Cloud DBs', author:'Patrick',
                keywords: ['cloud', 'storage'] },
{ title:'Graph Databases', author:'Larri', keywords: ['graph', 'NoSQL']}
)
```

HDFS stores unstructured log data from a scientific forum in text files where a single record corresponds to one post and contains a timestamp and username followed by a variable number of fields storing the keywords mentioned in the post:

`Posts (date, author, kw`$_1$`, kw`$_2$`, …, kw`$_n$`)`

```
2014-11-10, alice, storage, cloud
2014-11-10, bob, cloud, virtual, app
2014-11-10, alice, cloud
```

Query 1. This query aims at finding appropriate reviewers for publications of authors with a certain affiliation. It considers each publication's keywords and the experts

who have mentioned them most frequently on the scientific forum. The query combines data from the three data stores and can be expressed as follows.

```
scientists( name string, affiliation string )@rdb = (
  SELECT name, affiliation
  FROM scientists
)

publications(autor string, title string, keywords array)@mongo = (
  SELECT author, title, keywords
  FROM publications
)

experts(kw string, expert string)@hdfs = {*
  SCAN(TEXT, 'posts.txt', ',')                                          (op1)
    .FLAT_MAP( lambda data: product(data[2:], [data[1]]) )              (op2)
    .MAP( TUPLE, 1 )                                                    (op3)
    .REDUCE( SUM )                                                      (op4)
    .MAP( KEY[0], (KEY[1], VALUE) )                                     (op5)
    .REDUCE( lambda a, b: b if b[1] > a[1] else a )                     (op6)
    .PROJECT(KEY, VALUE[0])                                             (op7)
 *}

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

Query 1 contains three subqueries. The first two subqueries is a typical SQL statement to get data about respectively scientists (from PostgreSQL) and scientific publications (from MongoDB). The third subquery is an MFR operation chain that transforms the unstructured log data from the forum posts and represents the result of text analytics as a relation that maps each keyword to the person who has most frequently mentioned it. To achieve the result dataset, the MFR operations request transformations over the stored data, each of which is expressed either in a declarative way or with anonymous (lambda) Python functions.

The `SCAN` operation **`op1`** reads data from the specified text source and splits each line to an array of values. Let us recall that the produced array contains the author of the post in its second element and the mentioned keywords in the subarray starting from the third element. The following `FLAT_MAP` operation **`op2`** consumes each emitted array as a tuple and transforms each tuple using the defined Python lambda function, which performs a Cartesian product between the keywords subarray and the author, thus emitting a number of keyword-author pairs. Each of these pairs is passed to the `MAP` operation **`op3`**, which produces a new dataset, where each keyword-author pair is mapped to a value of 1. Then the `REDUCE` operation **`op4`** aggregates the number of occurrences for each keyword-author pair. The next `MAP` operation **`op5`** transforms the dataset by mapping each keyword to a pair of author-occurrences. The `REDUCE` **`op6`** finds for each keyword the author with the maximum number of occurrences, thus finding the expert who has mostly used the keyword. Finally, the `PROJECT` defines the mapping between the dataset fields and the columns of the returned relation.

Query Processing. First, Query 1 is compiled into the preliminary execution plan, depicted in Fig. 2. Then, the query optimizer finds the opportunity for pushing down the condition `affiliation = 'INRIA'` into the relational data store. Thus, the selection condition is included in the WHERE clause of the subquery for `rdb`. Doing this, the compiler determines that the column `s.affiliation` is no longer referenced in the common execution plan, so it is simply removed from the corresponding projection on `scientists` from `rdb`. This pushdown implies increasing the selectivity of the subquery, which is identified by the optimizer as an opportunity for performing a bind join. To further verify this opportunity, the query optimizer asks `rdb` to estimate the cardinality for the rewritten SQL subquery and, considering also the availability of an index on the field `author` in the MongoDB collection `publications`, the optimizer plans for bind join by pushing into the sub-plan for MongoDB the selection condition `author IN <authors>`, where `<authors>` refers to the list of distinct values of the `s.name` column, which will be determined at runtime.

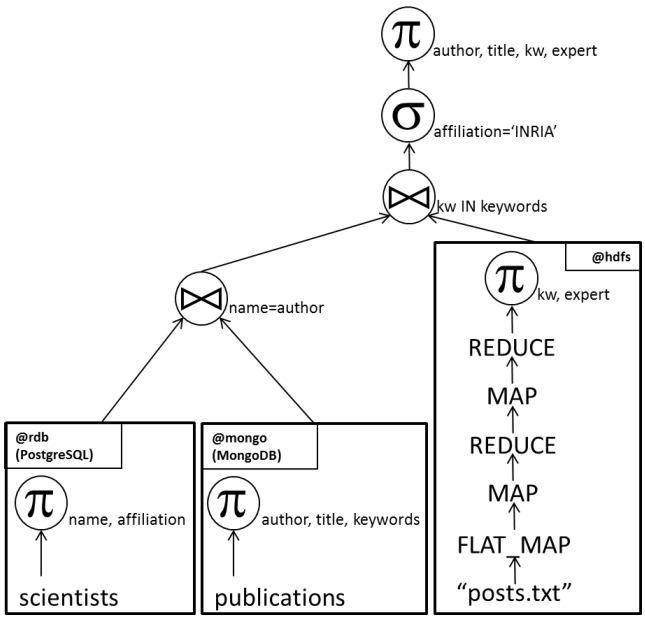


Fig. 2. Preliminary query plan for Query 1

Analogously, by using the catalog information provided by the MongoDB wrapper to estimate the cardinality of the join between scientists and publications, the optimizer plans to also involve the MFR subquery into a bind join and thus pushes the bind join condition `kw IN (<keywords>)`. Here, `<keywords>` is a placeholder for the list of distinct keywords retrieved from the column `p.keywords`. Recall that each value in `p.keywords` is an array, so the query processor will have to first flatten the intermediate relation by transforming the array-type column `p.keywords` to a scalar-

type column named `__keywords`. Since `p.keywords` participates in the join condition `kw IN keywords`, its flattening leads to transforming the join to an equi-join which allows for the query engine to utilize efficient methods for equi-joins.

Furthermore, the MFR planner seeks for opportunities to move the bind join filter condition `kw IN (<keywords>)` earlier in the MFR operation chain by applying the MFR rewrite rules, explained below. At this stage, although `<keywords>` is not known, the planner has all the information needed to apply the rules. After these transformations, the optimized query plan (Fig. 3) is executed by the query processor. In this notation, we use the symbol **F** to denote the flattening operator.

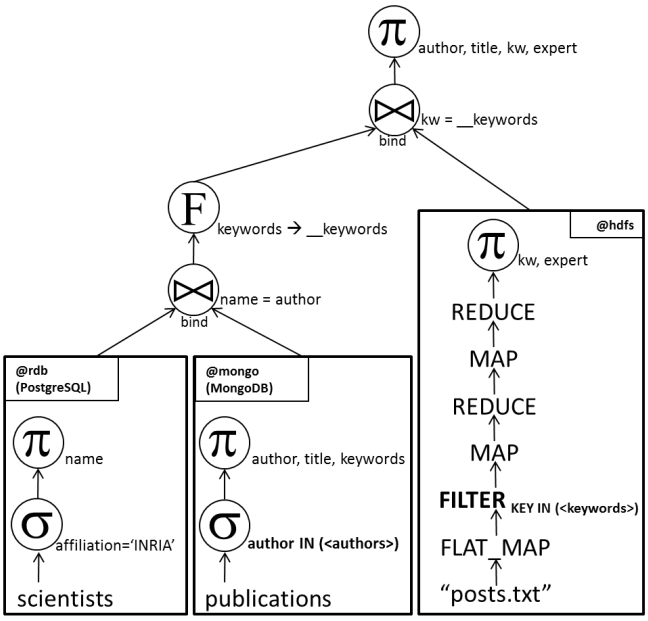


Fig. 3. Optimized query plan for Query 1

To execute the query plan, the query engine takes the following steps:

1. The query processor delivers to the wrapper of `rdb` the following SQL statement, rewritten by taking into account the pushed selection condition, for execution against the PostgreSQL data store, and waits for the corresponding result set to be retrieved in order to compose the bind join condition for the next step.

```
SELECT name
FROM scientists
WHERE affiliation = 'INRIA'
```

Name
Patrick
Boyan

2. The MongoDB wrapper prepares a native query to send to the MongoDB database to retrieve those tuples from `publications` that match the bind join criteria. It takes

into account the bind join condition derived from the already retrieved data from `rdb` and generates a MongoDB query whose SQL equivalent would be the following:

```
SELECT title, author, keywords FROM publications
WHERE author IN ('Patrick', 'Boyan')
```

However, the wrapper does not generate an SQL statement; instead it generates directly the corresponding MongoDB native query:

```
db.publications.find(
  { author: {$in:['Patrick', 'Boyan']} },
  { title: 1, author: 1, keywords: 1, _id: 0 }
)
```

Upon receiving the result dataset (a MongoDB document collection), the wrapper converts it to a table, according to the signature of the named table expression `publications`, ready to be joined with the already retrieved result set from step 1. The result of the bind join is the contents of the following intermediate relation:

author	title	keywords
Patrick	Principles of DDBS	['cloud', 'storage']

3. The flattening operator transforms the intermediate relation from step 2 to the following one:

author	title	__keywords
Patrick	Principles of DDBS	cloud
Patrick	Principles of DDBS	storage

4. The query processor identifies a list of the distinct values of the join attribute `__keywords` and derives from it the bind join condition `kw IN ('cloud', 'storage')` to push inside the subquery against `hdfs`.

5. The MFR planner for the wrapper of `hdfs` decides at which stage of the MFR sequence to insert the filter, by applying a number of rewrite rules. According to rule #1, the planner initially inserts the filter just before the `PROJECT` **`op7`** by rewriting the condition expression as follows:

```
.FILTER( KEY IN ('cloud', 'storage') )
```

Next, by applying consecutively rules #2 and #3, the planner moves the `FILTER` before the `MAP` **`op5`** by rewriting its condition expression according to rule #3:

```
.FILTER( KEY[0] IN ('cloud', 'storage') )
```

Analogously, rules #2 and #3 are applied again, moving the `FILTER` before **`op3`**, rewriting the expression once again, and thus settling it to its final position. After all transformations the MFR subquery is converted to the final MFR expression below.

```
 SCAN( TEXT, 'posts.txt', ',' )
.FLAT_MAP( lambda data: product(data[2:], [data[1]]) )
.FILTER( TUPLE[0] IN ('cloud', 'storage') )
.MAP( TUPLE, 1 )
.REDUCE( SUM )
.MAP( KEY[0], (KEY[1], VALUE) )
.REDUCE( lambda a, b: b if b[1] > a[1] else a )
```

6. The wrapper interprets the reordered MFR sequence, translates it to the Python script below as per the Python API methods of Spark, and executes it within the Spark framework.

```
 sc.textFile('posts.txt').map( lambda line: line.split(',') ) \
.flatMap( lambda data: product(data[2:], [data[1]]) ) \
.filter( lambda tup: tup[0] in ['cloud','storage'] ) \
.map( lambda tup: (tup, 1) ) \
.reduceByKey( lambda a, b: a + b ) \
.map( lambda tup: (tup[0][0], (tup[0][1], tup[1])) ) \
.reduceByKey( lambda a, b: b if b[1] > a[1] else a )
```

The result of MFR query reordering and interpreting on Spark is another intermediate relation:

kw	expert
cloud	alice
storage	alice

7. The intermediate relations from steps 3 and 6 are joined to produce the final result that lists the suggested experts for each publication regarding the given keywords:

author	title	kw	expert
Patrick	Principles of DDBS	cloud	alice
Patrick	Principles of DDBS	storage	alice

6 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. To achieve this, we have implemented a prototype of our query engine, aiming at implementing the proposed optimization techniques. In this section, we first describe the current implementation of the query engine prototype. Then, we introduce the datasets, based on the use case example in Section 5. Finally, we present our experimental results.

6.1 Prototype

For the purpose of our experiments, we have developed a prototype that invokes the Spark SQL [2] engine to perform data integration. The query compiler/optimizer is implemented in C++; it compiles a CloudMdsQL query into an optimized query execution plan. Then, a flow of invocations of Spark SQL's Python API methods is generated out of the execution plan. Thus, each MFR subquery, after being translated to a Python piece of code, is natively executed in the Spark context, while for performing relational operations on MFR and SQL named tables our prototype takes advantage of Spark SQL's DataFrame API. Wrappers are implemented as Python classes, whose `execute()` method accepts a native query or a query sub-plan, executes the corresponding query against its data store, and returns a DataFrame object ready to be con-

sumed by relational operators at Spark SQL. In our evaluation scenario, we use three data stores (`rdb`, `mongo`, and `hdfs`) whose wrappers are implemented as follows:

- The PostgreSQL wrapper loads a PostgreSQL data source by invoking `sqlContext.read().format("jdbc")`. Thus, the wrapper is able to execute SQL statements against the relational database using its JDBC driver. The wrapper exports an `explain()` function that the query optimizer invokes to get an estimation of the cost of a subquery. It can also be queried about the existence of certain indexes on table columns and their types.
- 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. It uses the `pymongo` library to query the database and then transforms a result set into a Spark DataFrame. The wrapper maintains the catalog information by running probing queries such as `db.collection.stats()` to keep actual database statistics. Similarly to the PostgreSQL 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.

Normally, if the QEP involves no bind joins, after all data frames that correspond to all named tables within a query are loaded into the Spark SQL context, the query engine simply invokes `sqlContext.sql()` to execute the integration `SELECT` statement as is. In case of a bind join, the query engine takes a couple of more steps. First, it performs a `SELECT DISTINCT` query on an intermediate table and then uses the retrieved distinct values to build the bind join condition that will be pushed inside the subquery for the other named table that participates in the join. If there is a flatten operator, the query engine uses the `LATERAL VIEW` clause available in Spark SQL. In our use case example, the publications named table is flattened into a temporary table using the command:

```
SELECT author, title, __keywords
FROM publications
LATERAL VIEW explode(keywords) _k AS __keywords
```

Then, to do the bind join, `SELECT DISTINCT __keywords` is performed on that temporary table.

6.2 Datasets

We performed our experimental evaluation in the context of the use case example, presented in Section 5. For this purpose, we generated data to populate the PostgreSQL table `scientists`, the MongoDB document collection `publications`, and

text files with unstructured log data stored in HDFS. All data is uniformly distributed and consistent. The datasets have the following characteristics:

- Table `scientists` contains 10K rows, distributed over 1000 distinct affiliations, making 10 authors per affiliation.
- Collection `publications` contains 10M documents, with uniform distribution of values of the `author` attribute, making 1K publications per scientist. Each publication is randomly assigned a set of 6 to 10 keywords out of 10K distinct `keyword` values. Also, there is an association between authors and keywords, so that all the publications of a single author reference only 1% of all the keywords. This means that a join involving the publications of a single author will have a selectivity factor of 1%; hence 100 distinct values for the bind join condition. The total size of the collection is 10GB.
- HDFS contains 16K files distributed between the nodes, with 100K tuples per file making 1.6 billion tuples, corresponding to posts from 10K forum users with 10K distinct keywords mentioned by them. The first field of each tuple is a timestamp and does not have an impact on the experimental results. The second field contains the author of the post as a string value. The remainder of the tuple line contains 1 to 10 `keyword` string values, randomly chosen out of the same set of 10K distinct keywords. The total size of the data is 124GB.

6.3 Experimental Results

To evaluate the impact of optimization on query execution, we use a cluster of the GRID5000 platform (www.grid5000.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, 64 GB main memory, and the network bandwidth is 10Gbps.

To demonstrate in detail the optimization techniques and their impact on the query execution, we prepared 3 different queries. We execute each of them 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. We do not focus on evaluating the bind join between PostgreSQL and MongoDB, as its benefit is less significant when compared to the benefit of doing bind join to the MFR subquery, because of the big difference in data sizes.

All the queries use the following common named table expressions, which we created as stored expressions:

```
CREATE NAMED EXPRESSION
scientists( name string, affiliation string )@rdb = (
  SELECT name, affiliation
  FROM scientists
);

CREATE NAMED EXPRESSION
publications(autor string, title string, keywords array)@mongo = (
```

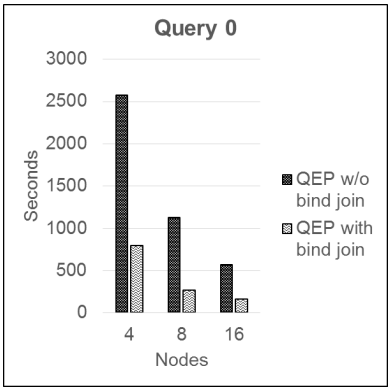
```
  SELECT author, title, keywords
  FROM publications
);

CREATE NAMED EXPRESSION
experts(kw string, expert string)@hdfs = {*
  SCAN(TEXT, 'posts.txt', ',')
    .FLAT_MAP( lambda data: product(data[2:], [data[1]]) )
    .MAP( TUPLE, 1 )
    .REDUCE( SUM )
    .MAP( KEY[0], (KEY[1], VALUE) )
    .REDUCE( lambda a, b: b if b[1] > a[1] else a )
    .PROJECT(KEY, VALUE[0])
 *};

CREATE NAMED EXPRESSION
experts_alt(kw string, expert string)@hdfs = {*
  SCAN(TEXT, 'posts.txt', ',')
    .FLAT_MAP( lambda data: product(data[2:], [data[1]]) )
    .MAP_VALUES(lambda v: Counter([v]))
    .REDUCE(lambda C1, C2: C1 + C2)
    .MAP_VALUES( lambda C: \
        reduce(lambda a,b: b if b[1] > a[1] else a, C.items()) )
    .PROJECT(KEY, VALUE[0])
 *};
```

Thus, each of the queries is expressed as a single SELECT statement that uses the above named table expressions. The named tables `scientists`, `publications`, and `experts` have exactly the same definition as in the use case example from Section 5.

The named table `experts_alt` does the same as `experts`, but its MFR sequence contains only one `REDUCE` (respectively, it does only one shuffle) and more complex map functions. It uses Python's Counter dictionary collection, with the additive property to sum up numeric values grouped by the key. The first `MAP_VALUES` maps a keyword to a Counter object, initialized with a single author key. Then the `REDUCE` sums all Counter objects associated to a single keyword, so that the result from it is an aggregated Counter dictionary, where an author is mapped to a number of occurrences of the keyword. The final `MAP_VALUES` uses Python's `reduce()` function (note that this is not Spark's reduce operator) to choose from all items in a Counter the author with the highest number of occurrences for a keyword.

Query 0 involves only the MongoDB database and the DPF to find experts for the publications of only one author. Thus, the selectivity factor of the bind join is 1%, as the number of keywords used by a single author is 1% of the total number of keywords. As we experimented with different number of nodes, we observe that the query execution efficiency and the benefit of the bind join scale well when the number of nodes increases. This is also observed in the rest of the queries.

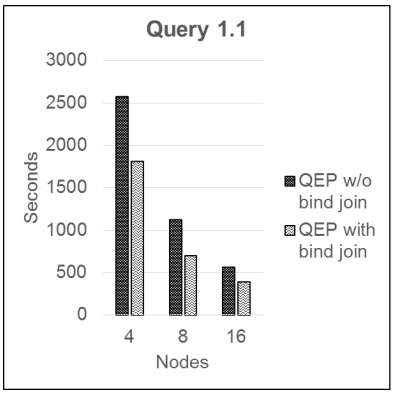
```
-- Query 0
SELECT p.author, p.title, e.kw, e.expert
FROM publications p, experts e
WHERE p.author = 'author1'
  AND e.kw IN p.keywords
```

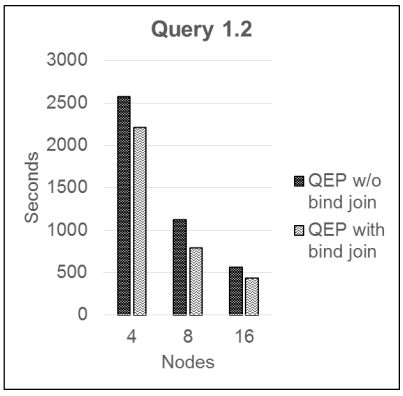


Query 1, as already introduced in Section 5, involves all the data stores and aims at finding experts for publications of authors with a certain affiliation. This makes a selectivity factor of 10% for the bind join, as there are 10 authors per affiliation. In addition, we explore another variant of the query, filtered to three affiliations, or 30% selectivity factor of the bind join. We enumerate the two variants as Query1.1 and Query 1.2.

```
-- Query 1.1: selectivity factor 10%
SELECT p.author, p.title, e.kw, e.expert
FROM scientists s, publications p, experts 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 e
WHERE s.affiliation IN ('affiliation1', 'affiliation2', 'affiliation3')
  AND p.author = s.name AND e.kw IN p.keywords
```

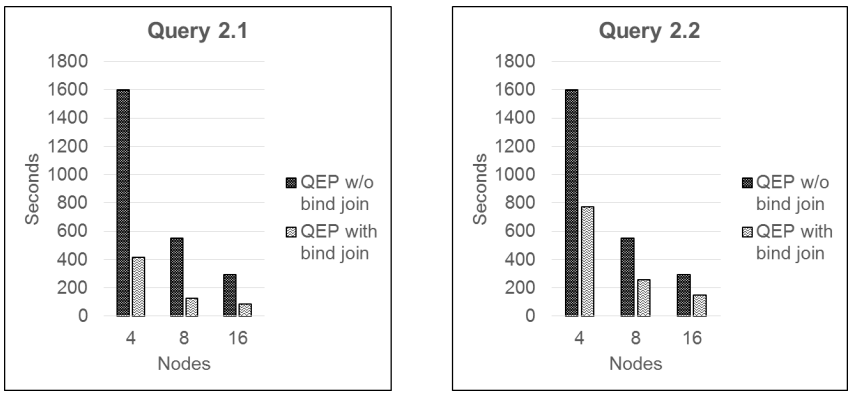




Query 2 does the same as Query 1, but uses the MFR subquery `experts_alt`, which uses more sophisticated map functions, but makes only one shuffle, where the key is a keyword. For comparison, the MFR expression `experts` makes two shuffles, of which the first one uses a bigger key, composed of a keyword-author pair. Therefore, the corresponding Spark computation of Query 2 involves much smaller size of data to be shuffled compared to Query 1, which explains its better overall efficiency and higher relative benefit of using bind join. Like with Query 1, we explore two variants with different selectivity factors of the bind join condition.

```
-- Query 2.1: 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 2.2: 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 Section 4.3).

7 Related Work

The problem of accessing heterogeneous data sources has long been studied in the context of multidatabase and data integration systems [16]. The typical solution is to provide a common data model and query language to transparently access data sources through a mediator, thus hiding data source heterogeneity and distribution.

The main requirements for a common query language (and data model) are support for nested queries, schema independence, and data-metadata transformation [22]. Nested queries allow queries to be arbitrarily chained together in sequences, so the result of one query (for one data store) may be used as the input of another (for another data store). Schema independence allows the user to formulate queries that are robust in front of schema evolution. Data-metadata transformation is important to deal with heterogeneous data models. To satisfy these requirements, several functional SQL-like languages have been introduced, with Functional SQL [20] being the first of them. More recently, FunSQL [3] has been proposed for the cloud, to allow shipping the code of an application to its data.

With respect to combining SQL and map/reduce operators, a number of SQL-like query languages have been recently introduced. HiveQL is the query language of the data warehousing solution Hive, built on top of Hadoop MapReduce [18]. Hive gives a relational view of HDFS stored unstructured data. HiveQL queries are decomposed to relational operators, which are then compiled to MapReduce jobs to be executed on Hadoop. In addition, HiveQL allows custom scripts, defining MapReduce jobs, to be referred in queries and used in combination with relational operators. SCOPE [6] is a declarative language from Microsoft designed to specify the processing of large sequential files stored in Cosmos, a distributed computing platform. SCOPE provides selection, join and aggregation operators and allows the users to implement their own operators and user-defined functions. SCOPE expressions and predicates are translated into C#. In addition, it allows implementing custom extractors, processors and reducers and combining operators for manipulating rowsets. SCOPE has been extended to combine SQL and MapReduce operators in a single language [24]. These systems are used over a single distributed storage system and therefore do not address the problem of integrating a number of diverse data stores.

To access heterogeneous databases, the mediator/wrapper architecture has several advantages. First, the specialized components of the architecture allow the various concerns of different kinds of users to be handled separately. Second, mediators typically specialize in a related set of data sources with “similar” data, and thus export schemas and semantics related to a particular domain. The specialization of the components leads to a flexible and extensible distributed system. In particular, it allows seamless integration of different data stored in very different data sources, ranging from full-fledged relational databases to simple files. DISCO [19] is a data integration system for accessing Web data sources, using an operator-based approach. It combines a generic cost model with specific cost information provided by the data source wrappers, thus allowing flexible cost estimation.

More recently, with the advent of cloud databases and big data processing frameworks, multidatabase solutions have evolved towards multistore systems that provide

integrated access to a number of RDBMS, NoSQL and HDFS data stores through a common query engine. We can divide multistore systems between loosely-coupled, tightly-coupled and hybrid.

Loosely-coupled multistore systems are reminiscent of multidatabase systems in that they can deal with autonomous data stores, which can then be accessed through the multistore system common interface as well as separately through their local API. Most loosely-coupled systems support only read-only queries. Loosely-coupled multistore systems follow the mediator/wrapper architecture with several data stores (e.g. NoSQL and RDBMS). BigIntegrator [14] integrates data from cloud-based NoSQL big data stores, such as Google's Bigtable, and relational databases. The system relies on mapping a limited set of relational operators to native queries expressed in GQL (Google Bigtable query language). With GQL, the task is achievable because it represents a subset of SQL. However, it only works for Bigtable-like systems and cannot integrate data from HDFS. QoX [17] integrates data from RDBMS and HDFS data stores through an XML common data model. It produces SQL statements for relational data stores, and Pig/Hive code for interfacing Hadoop to access HDFS data. The QoX optimizer uses a dataflow approach for optimizing queries over data stores, with a black box approach for cost modeling. SQL++ [15] mediates SQL and NoSQL data sources through a semi-structured common data model. The data model supports relational operators and to handle efficiently nested data, it also provides a flatten operator. The common query engine translates subqueries to native queries to be executed against data stores with or without schema. All these approaches mediate heterogeneous data stores through a single common data model. The polystore BigDAWG [9] goes one step further by defining "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. The system enables cross-island queries (across different data models) by moving intermediate datasets between islands in an optimized way.

Tightly-coupled multistore systems have been introduced with the goal of integrating Hadoop or Spark for big data analysis with traditional (parallel) RDBMSs. Tightly-coupled multistore systems trade autonomy for performance, typically in a shared-nothing cluster, taking advantage of massive parallelism. Odyssey [11] enables storing and querying data within HDFS and RDBMS, using opportunistic materialized views. MISO [13] is a method for tuning the physical design of a multistore system (Hive/HDFS and RDBMS), i.e. deciding in which data store the data should reside, in order to improve the performance of big data query processing. The intermediate results of query execution are treated as opportunistic materialized views, which can then be placed in the underlying stores to optimize the evaluation of subsequent queries. JEN [23] allows joining data from two data stores, HDFS and RDBMS, with parallel join algorithms, in particular, an efficient zigzag join algorithm, and techniques to minimize data movement. As the data size grows, executing the join on the HDFS side appears to be more efficient. Polybase [8] is a feature of Microsoft SQL Server Parallel Data Warehouse to access HDFS data using SQL. It allows HDFS data to be referenced through external PDW tables and joined with native PDW tables using SQL queries. HadoopDB [1] provides Hadoop MapReduce/HDFS access to

multiple single-node RDBMS servers (e.g. PostgreSQL or MySQL) deployed across a cluster, as in a shared-nothing parallel DBMS. It interfaces MapReduce with RDBMS through database connectors that execute SQL queries to return key-value pairs. Estocada [5] is a self-tuning multistore platform for providing access to datasets in native format while automatically placing fragments of the datasets across heterogeneous stores. For query optimization, Estocada combines both cost-based and rule-based approaches.

Hybrid systems support data source autonomy as in loosely-coupled systems, and exploit the local data source interface as in tightly-coupled systems, and typically HDFS through a parallel data processing framework like MapReduce or Spark. Spark SQL [2] is a parallel SQL engine built on top of Apache Spark and designed to provide tight integration between relational and procedural processing through a declarative API that integrates relational operators with procedural Spark code, taking advantage of massive parallelism. Spark SQL provides a DataFrame API that can map to relations arbitrary object collections and thus enables relational operations across Spark's RDDs and external data sources. In addition, it includes a flexible and extensible optimizer that supports operator pushdowns to data sources, according to their capabilities.

Our work fits in the hybrid system category as, similarly to Spark SQL, it uses Spark API to access the DPF data store, while querying the other stores through an SQL wrapper. However, it adds value by allowing the ad-hoc usage of user-defined map/reduce operators directly in MFR subqueries, yet allowing for optimization through the use of bind join and operator reordering. Furthermore, it does not give up the underlying data store's autonomy.

8 Conclusion

In this 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 query language can directly express subqueries that can take full advantage of the functionality of the underlying data stores and processing frameworks. Furthermore, it allows 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. Finally, compared with the related work on multistore systems, our work fits in the hybrid system category. However, it does not give up data store's autonomy, thus making our approach more general.

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. Finally, it allows for reducing the amount of processed data during the execution of the MFR sequence by reordering MFR operators according to

the determined rules. Our performance evaluation illustrates the query engine's ability to optimize a query and choose the most efficient execution strategy.

Acknowledgements

This research has been partially funded by the European Commission under project CoherentPaaS (FP7-611068).

References

1. 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, 922-933, 2009.
2. M. Armbrust, R. Xin, C. Lian, Y. Huai, D. Liu, J. Bradley, X. Meng, T. Kaftan, M. Franklin, A. Ghodsi, M. Zaharia. Spark SQL: Relational Data Processing in Spark. *ACM SIGMOD Int. Conf. on Management of Data*, 1383-1394, 2015.
3. C. Binnig, R. Rehrmann, F. Faerber, R. Riewe. FunSQL: It is time to make SQL functional. *EDBT/ICDT Conf.*, 41-46, 2012.
4. C. Bondiombouy, B. Kolev, O. Levchenko, P. Valduriez.: Integrating Big Data and Relational Data with a Functional SQL-like Query Language. *Int. Conf. on Databases and Expert Systems Applications (DEXA)*, 170-185, 2015.
5. F. Bugiotti, D. Bursztyn, A. Deutsch, I. Ileana, I. Manolescu. Invisible Glue: Scalable Self-Tuning Multi-Stores. *CIDR conf.*, 2015.
6. R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *PVLDB*, 1, 1265-1276, 2008.
7. CoherentPaaS project, http://coherentpaas.eu.
8. D. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, J. Gramling. Split Query Processing in Polybase. *ACM SIGMOD Conf.*, 1255-1266, 2013.
9. 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. *ACM SIGMOD Rec.* 44, 2 (August 2015), 11-16, 2015.
10. L. Haas, D. Kossmann, E. Wimmers, J. Yang. Optimizing Queries across Diverse Data Sources. *Int. Conf. on Very Large Databases (VLDB)*, 276-285, 1997.
11. H. Hacigümüs, J. Sankaranarayanan, J. Tatemura, J. LeFevre, N. Polyzotis. Odyssey: A Multi-Store System for Evolutionary Analytics. *PVLDB*, vol. 6, 1180-1181, 2013.
12. B. Kolev, P. Valduriez, C. Bondiombouy, R. Jiménez-Peris, R. Pau, J. Pereira. 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, 2015.
13. J. LeFevre, J. Sankaranarayanan, H. Hacigümüs, J. Tatemura, N. Polyzotis, M. Carey. MISO: souping up big data query processing with a multistore system. *ACM SIGMOD Conf.*, 1591-1602, 2014.
14. Z. Minpeng, R. Tore. Querying Combined Cloud-based and Relational Databases. *Int. Conf. on Cloud and Service Computing (CSC)*, 330-335, 2011.
15. 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.
16. T. Özsu, P. Valduriez. Principles of Distributed Database Systems. Springer, 2011.

17. A. Simitsis, K. Wilkinson, M. Castellanos, U. Dayal. Optimizing Analytic Data Flows for Multiple Execution Engines. *ACM SIGMOD Conf.*, 829-840, 2012.
18. A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, R. Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, vol. 2, 1626-1629, 2009.
19. A. Tomasic, L. Raschid, P. Valduriez. Scaling Access to Heterogeneous Data Sources with DISCO. *IEEE Trans. On Knowledge and Data Engineering*, vol. 10, 808-823, 1998.
20. P. Valduriez, S. Danforth. Functional SQL, an SQL Upward Compatible Database Programming Language. *Information Sciences*, vol. 62, 183-203, 1992.
21. G. Wiederhold. Mediators in the Architecture of Future Information Systems. *Computer*, vol. 25, 38-49, 1992.
22. C. M. Wyss, E.L. Robertson. Relational Languages for Metadata Integration. *ACM Trans. On Database Systems*, vol. 30(2), 624-660, 2005.
23. T. Yuanyuan, T. Zou, F. Özcan, R. Gonscalves, H. Pirahesh. Joins for Hybrid Warehouses: Exploiting Massive Parallelism in Hadoop and Enterprise Data Warehouses. *EDBT/ICDT Conf.*, 373-384, 2015.
24. J. Zhou, N. Bruno, M. Wu, P. Larson, R. Chaiken, D. Shakib. SCOPE: Parallel Databases Meet MapReduce. *PVLDB*, vol. 21, 611-636, 2012.
25. Q. Zhu, P.-A. Larson. A Query Sampling Method for Estimating Local Cost Parameters in a Multidatabase System. *Int. Conf. on Data Engineering (ICDE),* pp. 144-153, 1994.
26. Q. Zhu, P.-A. Larson. Global Query Processing and Optimization in the CORDS Multidatabase System. *Int. Conf. on Parallel and Distributed Computing Systems*, 640-647, 1996.
27. Q. Zhu, Y. Sun, S. Motheramgari. Developing Cost Models with Qualitative Variables for Dynamic Multidatabase Environments. *Int. Conf. on Data Engineering (ICDE)*, 413-424, 2000.