Parallel Polyglot Query Processing on Heterogeneous Cloud Data Stores with LeanXcale

Boyan Kolev, Oleksandra Levchenko, Esther Pacitti, Patrick Valduriez
Inria & LIRMM, University of Montpellier
Montpellier, France
e-mail: firstname.lastname@inria.fr

Ricardo Vilaça[1], Rui Gonçalves[1], Ricardo Jiménez-Peris[1], Pavlos Kranas[1,2]
[1] LeanXcale
[2] Distributed Systems Lab UPM
Madrid, Spain
{ricardo.vilaca,rui.goncalves,rjimenez,pavlos}@leanxcale.com

***Abstract*—The blooming of different cloud data stores has turned polystore systems to a major topic in the nowadays cloud landscape. Especially, as the amount of processed data grows rapidly each year, much attention is being paid on taking advantage of the parallel processing capabilities of the underlying data stores. To provide data federation, a typical polystore solution defines a common data model and query language with translations to API calls or queries to each data store. However, this may lead to losing important querying capabilities. The polyglot approach of the CloudMdsQL query language allows data store native queries to be expressed as inline scripts and combined with regular SQL statements in ad-hoc integration queries. Moreover, efficient optimization techniques, such as bind join, can still take place to improve the performance of selective joins. In this paper, we introduce the distributed architecture of the LeanXcale query engine that processes polyglot queries in the CloudMdsQL query language, yet allowing native scripts to be handled in parallel at data store shards, so that efficient and scalable parallel joins take place at the query engine level. The experimental evaluation of the LeanXcale parallel query engine on various join queries illustrates well the performance benefits of exploiting the parallelism of the underlying data management technologies in combination with the high expressivity provided by their scripting/querying frameworks.**

Keywords- polystores; cloud computing; query processing

I. INTRODUCTION

A major trend in cloud computing and data management is the understanding that there is no "one size fits all" solution [22]. Thus, there has been a blooming of different NoSQL cloud data management infrastructures, distributed file systems (e.g. Hadoop HDFS), and big data processing frameworks (e.g. Hadoop MapReduce, Apache Spark, or Apache Flink), specialized for different kinds of data and tasks and able to scale and perform orders of magnitude better than traditional relational DBMS. This has resulted in a rich offering of services that can be used to build cloud data-intensive applications that can scale and exhibit high performance. However, this has also led to a wide diversification of DBMS interfaces and the loss of a common programming paradigm, which makes it very hard for a user to efficiently integrate and analyze her data sitting in different data stores.

For example, let us consider a banking institution that keeps its operational data in a SQL database, but stores data about bank transactions in a document database, because each record typically contains data in just a few fields, so they make use of the semi-structured nature of documents. And because of the big volumes of data, both databases are sharded into multiple nodes in a cluster. On the other hand, a web application appends data to a big log file, stored in HDFS. In this context, an analytical query that involves datasets from both databases and the HDFS file would face three major challenges. First, in order to execute efficiently, the query needs to be processed in parallel, taking advantage of parallel join algorithms. Second, in order to do this, the query engine must be able to retrieve in parallel the partitions from the underlying data stores. And third, the query needs to be expressive enough, so as to combine an SQL subquery (to the relational database or the HDFS log file through an SQL engine, e.g. Hive) with an arbitrary code in a scripting language (e.g. JavaScript) that requests a dataset from the document database. Existing polystore solutions provide SQL mappings to document collections. However, this leads to limitations of important querying capabilities, as the underlying schema may be very far from relational and data transformations need to take place before being involved in relational operations. Therefore, we rather focus our work on leveraging the underlying data stores' scripting (querying) mechanisms.

A number of polystores that have been recently proposed partially address our problem. In general, they provide integrated access to multiple, heterogeneous data stores through a single query engine. Loosely-coupled polystores [5,9,10,17,18,21,25] typically respect the autonomy of the underlying data stores and rely on a mediator/wrapper approach to provide mappings between a common data model / query language and each particular data store's data model. CloudMdsQL [13,15] even allows data store native queries to be expressed as inline scripts and combined with regular SQL statements in ad-hoc integration queries. However, even when they access parallel data stores, loosely-coupled polystores typically do centralized access, and thus cannot exploit parallelism for performance. Another family of polystore systems [1,8,11,16,26] uses a tightly-coupled approach in order to trade data store autonomy and query expressivity for performance. 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 (parallel) data warehouse. Thus, tightly-coupled systems take advantage of massive parallelism by bringing in parallel shards from HDFS tables to the SQL database nodes and doing parallel joins. But they are limited to accessing only specific data stores, usually with SQL mappings of the data stores' query interfaces. However, according to a recent benchmarking [14], using native queries directly at the data store yields a significant performance improvement compared to mapping native datasets and functions to relational tables and operators. Therefore, what we want to provide is a hybrid system that combines high expressivity (through the use of native queries) with massive parallelism and optimizability.

In this paper, we present a query engine that addresses the afore-mentioned challenges of parallel multistore query processing. To preserve the expressivity of the underlying data stores' query/scripting languages, we use the polyglot approach provided by the CloudMdsQL query language, which also enables the use of bind joins to optimize the execution of selective queries. And to enable the parallel query processing, we incorporated the polyglot approach within the LeanXcale[1] Distributed Query Engine (DQE), which provides a scalable database that operates over a standard SQL interface.

The rest of this paper is organized as follows. Section 2 gives an overview of the query language and its polyglot capabilities. Section 3 discusses the distributed architecture of LeanXcale query engine. Our major contribution is presented in Section 4, where we describe the architectural extensions that turn the DQE into a parallel polyglot polystore system. Section 5 presents the experimental evaluation of various parallel join queries across data stores using combined SQL and native queries. Section 6 discusses related work. Section 7 concludes.

II. 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.

A. Query Language

The design of the query language is based on the assumption that the programmer has deep expertise and knowledge about the specifics of the underlying data stores, as well as awareness about how data are organized across them. Queries that integrate data from several data stores usually consist of native 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. 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 = {*
  return db.A.find( {x: {$lt: 10}}, {x:1, z:1} );
*}
SELECT T1.x, T2.z FROM T1, T2
WHERE T1.x = T2.x AND T1.y <= 3
```

The two subqueries are sent independently for execution against their data stores in order the retrieved relations to be joined at query engine level. The SQL table expression `T1` is defined by an SQL subquery, while `T2` is a native expression (identified by the special bracket symbols `{* *}`) expressed as a native MongoDB call. The subquery of expression `T1` is subject to rewriting by pushing into it the filter condition `y <= 3`, to increase efficiency.

CloudMdsQL allows named table expressions to be defined as functions in a scripting language (e.g., Python, JavaScript), which is useful for querying data stores that have only API-based query interface. A scripting expression can either yield tuples to its result set (like a user-defined table function) or return an iterable object that represents the result set (like in the MongoDB example above).

B. Bind Join

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 equi-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 $b_1 \ldots b_n$, 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 queries is through the use of a `JOINED ON` clause in the named table signature, like in the named table `A` below, defined as a MongoDB script.

[1] http://www.leanxcale.com

```
A(id int, x int JOINED ON id
    REFERENCING OUTER AS b_keys)@mongo =
{*  return db.A.find( {id: {$in: b_keys}} );  *}
```

Thus, when `A.id` participates in an equi-join, the values $b_1,\ldots,b_n$ are provided to the script code through the iterator/list object `b_keys` (in this context, we refer to the table `B` as the "outer" table, and `b_keys` as the outer keys).

III. LEANXCALE ARCHITECTURE OVERVIEW

The LeanXcale database has derived its OLAP query engine from Apache Derby, a Java-based open-source SQL database. Apache Derby is a centralized OLTP database. LeanXcale database is a scalable distributed Full ACID Full SQL database with OLTP and OLAP support. LeanXcale has three main subsystems: the query engine, the transactional engine and the storage engine, all three distributed and highly scalable (i.e. to 100s of nodes). The query engine is a distributed MPP engine that process OLAP workloads over the operational data, so that analytical queries are answered over real-time data. LeanXcale, thus, enables to avoid ETL processes to migrate data from operational databases to data warehouses by providing both functionalities in a single database manager. The parallel implementation of the query engine for OLAP queries follows the single-program multiple data (SPMD) approach [6], where multiple symmetric workers (threads) on different query instances execute the same query/operator, but each of them deals with different portions of the data. In this section we provide a brief overview of the query engine distributed architecture.

Figure 1 illustrates the architecture of LeanXcale's Distributed Query Engine (DQE). Applications connect to one of the multiple DQE instance running, which exposes a typical JDBC interface to the applications, with support for SQL and transactions. The DQE executes the applications' requests, handling transaction control, and updating data, if necessary. The data itself is stored on a proprietary relational key-value store, KiVi, which allows for efficient horizontal partitioning of LeanXcale tables and indexes, based on the primary key or index key. Each table is stored as a KiVi table, where the key corresponds to the primary key of the LeanXcale table and all the columns are stored as they are into KiVi columns. Indexes are also stored as KiVi tables, where the index keys are mapped to the corresponding primary keys. This model enables high scalability of the storage layer by partitioning tables and indexes across KiVi Data Servers (KVDS).

This architecture scales by allowing analytical queries to execute in parallel, in this way supporting **intra-query and intra-operator parallelism**. For parallel query execution, the initial connection (which creates the master worker) will start additional connections (workers), all of which will cooperate on the execution of the queries received by the master.

When a parallel connection is started, the master worker starts by determining the available DQE instances, and it decides how many workers will be created on each instance. For each additional worker needed, the master then creates a thread, which initiates a JDBC connection to the worker. Each JDBC connection is initialized as a worker, creating a communication end-point for an overlay network to be used for intra-query synchronization and data exchange. After the initialization of all workers the overlay network is connected. After this point, the master is ready to accept queries to process.

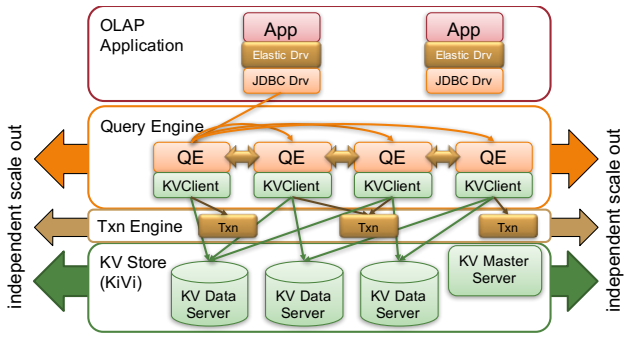


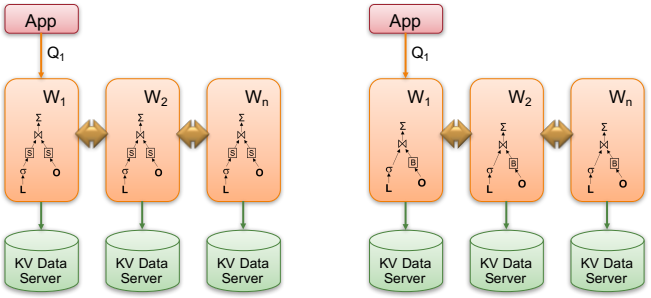
Figure 1. DQE distributed architecture.

As queries are received, query plans are broadcast and processed by all workers. For parallel execution, an optimization step is added, which transforms the generated sequential query plan into a parallel one. This transformation involves replacing table scans with parallel table scans, and adding shuffle operators to make sure that, in stateful operators (such as Group By, or Join), related rows are handled by the same worker. Parallel table scans will divide the rows from the base tables among all workers, i.e., each worker will retrieve a disjoint subset of the rows during table scans. This is done by dividing the rows and scheduling the obtained subsets to the different DQE instances. Each worker then processes the rows obtained from subsets scheduled to its DQE instance, exchanging rows with other workers as determined by the shuffle operators added to the query plan.

Let us consider the query Q_1 below, which we will use as a running example throughout the paper to illustrate the different query processing modes. The query assumes a TPC-H [24] schema.

```
Q1: SELECT count(*)
    FROM LINEITEM L, ORDERS O
    WHERE L_ORDERKEY = O_ORDERKEY
    AND L_QUANTITY < 5
```

This query is parsed into a query execution plan, where leaf nodes correspond to tables or index scans. The master worker then broadcasts to all workers the generated query plan, with the additional shuffle operators (Figure 2a). Then, the DQE scheduler assigns evenly all database shards across all workers. To handle the leaf nodes of the query plan, each worker will do table/index scans only at the assigned shards. Let us assume for simplicity that the DQE launches the same number of workers as KVDS servers, so each worker connects to exactly one KVDS server and reads the partition of each table that is located in that KVDS server. Then workers execute in parallel the same copy of the query plan, exchanging rows across each other at the shuffle operators (marked with an S box).



(a) with shuffles *(b) with broadcasts*

Figure 2. Query processing in parallel mode.

To process joins, the query engine may use different strategies. First, to exchange data across workers, **shuffle** or **broadcast** methods can be used. The shuffle method is efficient when both sides of a join are quite big; however, if one of the sides is relatively small, the optimizer may decide to use the broadcast approach, so that each worker has a full copy of the small table, which is to be joined with the local partition of the other table, thus avoiding the shuffling of rows from the large table (Figure 2b). Apart from the data exchange operators, the DQE supports various join methods (hash, nested loop, etc.), performed locally at each worker after the data exchange takes place.

IV. Parallel Polyglot Query Processing across Data Stores

LeanXcale DQE is designed to integrate with arbitrary data management clusters, where data resides in its natural format and can be retrieved (in parallel) by running specific scripts or declarative queries. These can range from distributed raw data files, through parallel SQL databases, to sharded NoSQL databases (such as MongoDB, where queries can be expressed as JavaScript programs). This turns LeanXcale DQE into a powerful "big data lake" polyglot query engine that can process data from its original format, taking full advantage of both expressive scripting and massive parallelism. Moreover, joins across any native datasets, including LeanXcale tables, can be applied, exploiting efficient parallel join algorithms. Here we specifically focus on parallel joins between a relational table and the result of a JavaScript subquery to MongoDB, but the concept relies on an API that allows its generalization to other script engines and data stores as well. To enable ad-hoc querying of an arbitrary data set, using its scripting mechanism, and then joining the retrieved result set at DQE level, DQE processes queries in the CloudMdsQL query language, where scripts are wrapped as native subqueries.

To better illustrate the necessity of enabling user-defined scripts to MongoDB as subqueries, rather than defining SQL mappings to document collections, let us consider the following MongoDB collection `orders` that has a highly non-relational structure:

```
{order_id: 1, customer: "ACME", status: "O",
 items: [
  {type: "book", title: "Book1", author: "A.Z."},
  {type: "phone", brand: "Samsung", os: "Android"}
] }, ...
```

Each record contains an array of item objects whose properties differ depending on the item type. A query that needs to return a table listing the title and author of all books ordered by a given customer, would be defined by means of a flatMap operator in JavaScript, following a MongoDB find() operator. The example below wraps such a subquery as a CloudMdsQL named table:

```
BookOrders(title string, author string)@mongo = {*
  return db.orders.find({customer: "ACME"})
  .flatMap( function(v) {
    var r = [];
    v.items.forEach( function(i){
      if (i.type == "book")
        r.push({title:i.title, author:i.author});
    } );
    return r; });
*}
```

And if this table has to be joined with a LeanXcale table named `authors`, this can be expressed directly in the main SELECT statement of the CloudMdsQL query:

```
SELECT B.title, B.author, A.nationality
FROM BookOrders B, Authors A
WHERE B.author = A.name
```

Furthermore, we aim at processing this join in the most efficient way, i.e. in parallel, by allowing parallel handling of the MongoDB subquery and parallel retrieval of its result set.

By processing such queries, DQE takes advantage of the expressivity of each local scripting mechanism, yet allowing for results of subqueries to be handled in parallel by DQE and involved in operators that utilize the intra-query parallelism. The query engine architecture is therefore extended to access in parallel shards of the external data store through the use of DataLake distributed wrappers that hide the complexity of the underlying data stores' query/scripting languages and encapsulate their interfaces under a common DataLake API to be interfaced by the query engine.

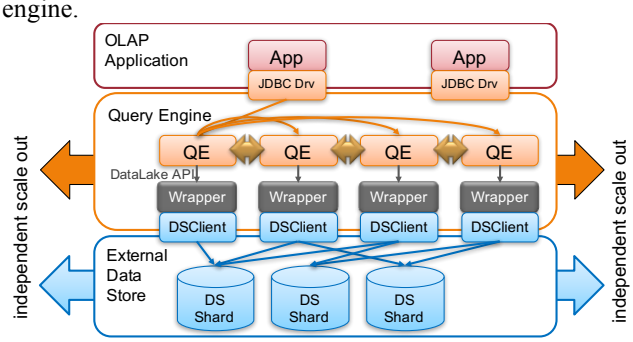


Figure 3. Generic architecture extension for accessing external data stores.

A. DataLake API

For a particular data store, each DQE worker creates an instance of the DataLake wrapper that is generally used for querying and retrieval of shards of data. Each wrapper typically uses the client API of the corresponding data management cluster and implements the following DataLake

API methods to be invoked by the query engine in order to provide parallel retrieval of shards (Figure 3).

The method *init(ScriptContext)* requests the execution of a script to retrieve data from the data store. It provides connection details to address the data store and the script as text. It may also provide parameter values, if the corresponding named table is parameterized. Normally, the wrapper does not initiate the execution of the script before a shard is assigned by the *setShard* method (see below).

After the initialization, the DQE selects one of the wrapper instances (the one created by the master worker) as a master wrapper instance. The method *Object[] listShards()* is invoked by the DQE only to the master wrapper to provide a list of shards where the result set should be retrieved from. Each of the returned objects encapsulates information about a single shard, which is implementation-specific, therefore transparent for the query engine. Such an entry may contain, for example, the network address of the database shard, and possibly a range of values of the partition key handled by this shard. Since the query engine is unaware of the structure of these objects, the wrapper provides additional methods for serializing and deserializing shard entries, so that DQE can exchange them across workers.

Having obtained all the available shards, the DQE schedules the shard assignment across workers and invokes the method *setShard(Object shard)* to assign a shard to a particular wrapper instance. Normally, this is the point where the connection to the data store shard takes place and the script execution is initiated. This method might be invoked multiple times to a single wrapper instance, in case there are more shards than workers.

Using the method *boolean next(Object[] row)*, the query engine iterates through a partition of the result set, which is retrieved from the assigned shard. When this iteration is over, the DQE may assign another shard to the wrapper instance.

By interfacing wrappers through the DataLake API, the DQE has the possibility to retrieve in parallel disjoint subsets of the result set, much like it does with LeanXcale tables. A typical wrapper implementation should use a scripting engine and/or a client library to execute scripts (client- or server-side) against the data store.

B. Implementation for MongoDB

In this section, we introduce the design of the distributed MongoDB wrapper. The concept of parallel querying against a MongoDB cluster is built on the assumption that each DQE worker can access directly a MongoDB shard, bypassing the MongoDB router in order to sustain parallelism. This, however, forces the DQE to define certain constraints for parallel processing of document collection subqueries, in order to guarantee consistent results, which is normally guaranteed by the MongoDB router. The full scripting functionality of MongoDB JavaScript library is still provided, but in case parallel execution constraints fail, the execution falls back to a sequential one. First, the wrapper verifies that the MongoDB balancer is not running in background, because otherwise it may be moving chunks of data across MongoDB shards at the same time the query is being executed, which may result in inconsistent reads. For an optimal operation of the parallel analytics engine, for example, the database administrator may schedule the balancer to be active only in not intensive for the analytics engine hours. Second, the subquery should use only stateless operators (`Op`) on document collections, as they are distributive over the union operator. In other words, for any disjoint subsets (shards) S_1 and S_2 of a document collection C, $Op(S_1) \cup Op(S_2) = Op(S_1 \cup S_2)$ must hold, so that the operator execution can be parallelized over the shards of a document collection while preserving the consistency of the resulting dataset. In our current work, we specifically focus on enabling the parallel execution of filtering, projection (map), and flattening operators with user-defined as JavaScript functions transformations.

The distributed wrapper for MongoDB comprises a number of instances of a Java class that implements the DataLake API, each of which embeds a JavaScript scripting engine that uses MongoDB's JavaScript client library. To support parallel data retrieval, we further enhance the client library with JavaScript primitives that wrap standard MongoCursor objects (usually returned by a MongoDB JavaScript query) in ShardedCursor objects, which are aware of the sharding of the underlying dataset. In fact, ShardedCursor implements all DataLake API methods and hence serves as a proxy of the API into the JavaScript MongoDB client library. The client library is therefore extended with the following document collection methods that return ShardedCursor and provide the targeted operators (find, map, and flat map) in user scripts.

The `findSharded()` method accepts the same arguments as the native MongoDB `find()` operator, in order to provide the native flexible querying functionality, complemented with the ability to handle parallel iteration on the sharded result set. Note that, as opposed to the behavior of the original `find()` method, a call to `findSharded()` does not immediately initiate the MongoDB subquery execution, but only memorizes the filter condition (the method argument), if any, in the returned ShardedCursor object. This delayed iteration approach allows the DQE to internally manipulate the cursor object before the actual iteration takes place, e.g. to redirect the subquery execution to a specific MongoDB shard. And since an instance of ShardedCursor is created at every worker, this allows for the parallel assignment of different shards.

In order to make a document result set fit the relational schema required by a CloudMdsQL query, the user script can further take advantage of the `map()` and `flatMap()` operators. Each of them accepts as argument a JavaScript mapper function that performs a transformation on each document of the result set and returns another document (map) or a list of documents (flatMap). Thus, a composition of `findSharded` and `map/flatMap` (such as in the `BookOrders` example above) makes a user script expressive enough, so as to request a specific MongoDB dataset, retrieve the result set in parallel, and transform it in order to fit the named table signature and further be consumed by relational operators at the DQE level.

Let us consider the following modification Q_1^{ML} of query Q_1, which assumes that the `LINEITEM` table resides as a sharded document collection in a MongoDB cluster and the selection on it is expressed by means of the `findSharded()` JavaScript method, while ORDERS is still a LeanXcale table, the partitions of which are stored in the KV storage layer.

```
Q1ML: LINEITEM( L_ORDERKEY int, … )@mongo = {*
      return db.lineitem.findSharded(
          {l_quantity: {$lt: 5}} );
  *}

  SELECT count(*)
  FROM LINEITEM L, ORDERS O
  WHERE L_ORDERKEY = O_ORDERKEY
```

Let us assume for simplicity the same number of DQE workers, KVDS servers, and MongoDB shards, so each worker gets exactly one partition of both tables by connecting to one MongoDB shard (through a wrapper instance) and one KVDS (Figure 4).

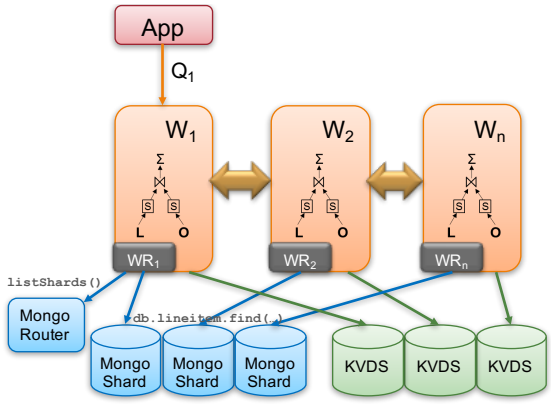


Figure 4. Parallel join between sharded datasets: LeanXcale table and MongoDB collection.

The DQE initiates the subquery request by passing the script code to each wrapper instance through a call to its `init()` method. At this point, the ShardedCursor object does not yet initiate the query execution, but only memorizes the query filter object. Assuming that W_1 is the master worker, it calls the `listShards()` method of its wrapper instance WR_1 to query the MongoDB router for a list of MongoDB shards (database instances identified by host address and port), where partitions of the `lineitem` collection are stored. The list of shards is then reported to the DQE scheduler, which assigns one MongoDB shard to each of the workers by calling the `setShard()` method. Each worker then connects to the assigned shard and invokes the `find()` method to a partition of the `lineitem` collection using the memorized query condition, thus retrieving a partition of the resulting dataset (if a `flatMap` or `map` follows, it is processed for each document of that partition locally at the wrapper). The dataset partition is then converted to a partition of an intermediate relation, according to the signature of the `LINEITEM` named table expression. At this point, the DQE is ready to involve the partitioned intermediate relation `LINEITEM` in the execution of a parallel join with the native LeanXcale partitioned table `ORDERS`.

C. *Implementation for HDFS Files*

The distributed HDFS wrapper is designed to access in parallel tables stored as HDFS files, thus providing the typical functionality of a tightly-coupled polystore, but through the use of the DataLake API. We assume that each accessed HDFS file is registered as table in a Hive metastore. Therefore, a wrapper instance can use the Hive metastore API to get schema and partitioning information for the subqueried HDFS table and hence to enable iteration on a particular split (shard) of the table. Note that Hive is interfaced only for getting metadata, while the data rows are read directly from HDFS. To better illustrate the flow, let us consider another modification Q_1^{HL} of query Q_1, which assumes that the `LINEITEM` table is stored as file in a Hadoop cluster.

```
Q1HL: SELECT count(*)
      FROM LINEITEM@hdfs L, ORDERS O
      WHERE L_ORDERKEY = O_ORDERKEY
```

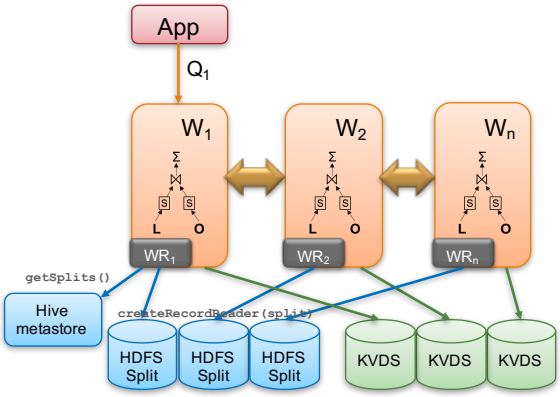


Figure 5. Parallel join between LeanXcale and HDFS tables.

To schedule parallel retrieval of the `LINEITEM` table, the DQE redirects the subquery to the HDFS wrapper, preliminarily configured to associate the `@hdfs` alias with the URI of the Hive metastore, which specifies how the file is parsed and split. This information is used by the master wrapper, which reports the list of file splits (instances of Hive API's `InputSplit` class) to the DQE scheduler upon a call to the `listShards()` method. Then, the scheduler assigns a split to each of the workers, which creates a record reader on it in order to iterate through the split's rows (Figure 5).

V. Experimental Evaluation

The goal of our experimental validation is to assess the scalability of the query engine when processing integration (join) queries across diverse data sources, as our major objective is to be able to fully exploit both the massive parallelism and high expressivity, provided by the underlying data management technologies and their scripting frameworks. We evaluate the scalability of processing a

particular query by varying the volume of queried data and the level of parallelism and analyzing the corresponding execution times. In particular, we strive to retain similar execution times of a particular query when keeping the level of parallelism (in number of data shards and workers) proportional to the scale of data.

The experimental evaluation was performed on a cluster of the GRID5000 platform[2]. Each node in the cluster runs on two Xeon E5-2630 v3 CPUs at 2.4GHz, 16 logical cores per CPU (i.e. 32 per node), 128 GB main memory, and the network bandwidth is 10Gbps. The highest level of parallelism is determined by the total number of cores in the cluster. We performed the experiments varying the number of nodes from 2 to 16 and the number of workers from 32 to 512 (several workers per node). All the three data stores and the query engine are evenly distributed across all the nodes, i.e. shards of each data store are collocated at each node. For each experiment, the level of parallelism determines the number of data shards, as well as the highest number of workers, in accordance with the total number of cores in the cluster. As the number of nodes did not show significance compared to the number of workers, our experimental conclusions refer only to the number of workers.

We performed our experiments in three general groups of test cases, each having a distinct objective. All the queries were run on a cluster of LeanXcale DQE instances, running the distributed wrappers for MongoDB and Hive. For comparison with the state of the art, the large scale test case queries were also performed on a Spark SQL cluster, where we used the MongoDB Spark connector to access MongoDB shards in parallel.

A. *Generic Scalability*

The first group of test cases aims at generic evaluation of the performance and scalability of joins across any pair of the three involved data stores. The data used was based on the TPC-H benchmark schema [24], particularly for the tables LINEITEM, ORDERS, and CUSTOMER. All the generated datasets were: loaded in LeanXcale as tables; loaded in MongoDB as document collections; copied to the HDFS cluster as raw CSV files, to be accessed through Hive as tables. To perform the tests on different volumes of data, the datasets were generated with three different scale factors – 60GB, 120GB, and 240GB. Note that here we focus just on the evaluation of joins; therefore, our queries involve only joins over full scans of the datasets, without any filters.

The six queries used for this evaluation are variants of the following:

```
Q1: SELECT count(*)
    FROM LINEITEM L, ORDERS O
    WHERE L_ORDERKEY = O_ORDERKEY
```

We will refer to them with the notation Q_1^{XY}, where X is the first letter of the data store, from which LINEITEM is retrieved, while Y refers to the location of ORDERS. For example, Q_1^{ML} joins LINEITEM from MongoDB with ORDERS from LeanXcale. Subqueries to MongoDB are expressed natively in JavaScript and intermediate result sets from MongoDB and HDFS retrieved in parallel, as described in Section 4.

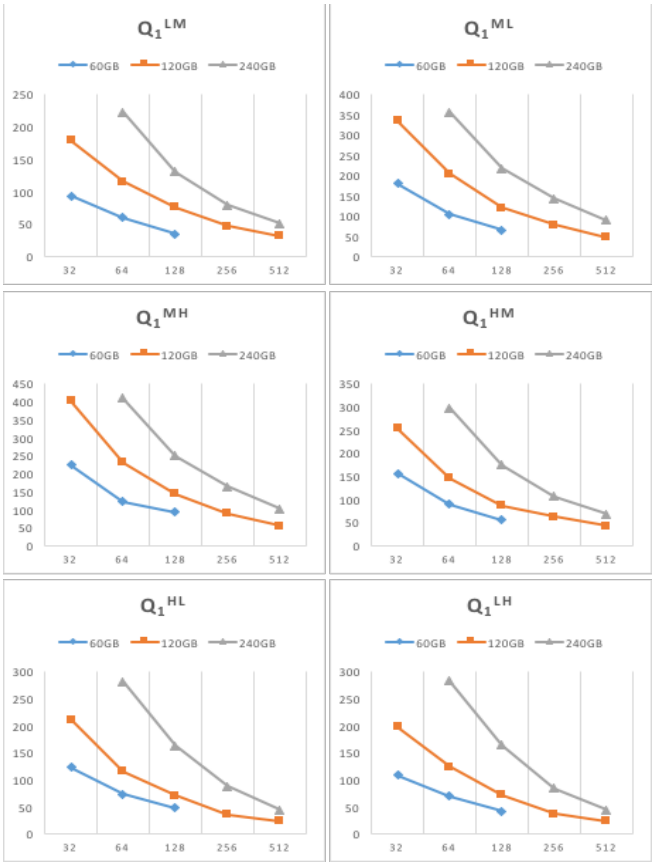


Figure 6. Execution times (in seconds) of Q_1 queries on TPC-H data with different scales of data (60, 120, and 240 GB) and different levels of parallelism (32, 64, 128, 256, and 512 workers).

Figure 6 shows the performance measurements on queries of the first test case, executing joins between LINEITEM and ORDERS tables in any configuration of pairs between the three data stores.

In general, the execution speed is determined by the performance of processing the LINEITEM side of the join, as this table is much larger than ORDERS. When LINEITEM resides at LeanXcale, the performance is highest, as the query engine processes it natively. For HDFS tables, some overhead is added, due to data conversions, communication with the Hive metastore, and possibly accessing HDFS splits through the network. MongoDB subqueries show lowest performance as data retrieval passes through the embedded at each worker JavaScript interpreter.

All the graphs show reasonable speedup with increase of the parallelism level. Moreover, the correspondence between scale of data and parallelism level is quite stable. For example, quite similar execution times are observed for 60GB with 64 workers, 120GB with 128 workers, and 240GB with 256 workers. This means that, as the volume of data grows, performance can be maintained by simply adding a proportional number of workers and data shards.

[2] http://www.grid5000.fr

B. *High Expressivity and Scalability*

The second test case aims at the evaluation of highly expressive JavaScript subqueries, such as the `BookOrders` example from Section 4. The goal is to show that even with more sophisticated subqueries, scalability is not compromised. For the purpose, we created a MongoDB nested document collection named `Orders_Items`, where we combined the ORDERS and LINEITEM datasets as follows. For each ORDERS row we created a document that contains an additional array field `items`, where the corresponding LINEITEM rows were added as subdocuments. Each of the item subdocuments was assigned a `type` field, the value of which was randomly chosen between "book" and "phone". Then, "title" and "author" fields were added for the "book" items and "brand" and "os" – for the "phone" items, all filled with randomly generated string values. Thus, the following `BookOrders` named table was used in the test queries:

```
BookOrders(custkey int, orderdate date,
           title string, author string)@mongo =
{*
  return db.orders_items.findSharded()
  .flatMap( function(doc) {
    var r = [];
    doc.items.forEach( function(i){
      if (i.type == "book")
        r.push({custkey: doc.custkey,
                orderdate: doc.orderdate,
               title: i.title, author: i.author});
    } );
    return r; });
*}
```

We ran two queries under the same variety of conditions – three different scale factors for the volume of data and varying the level of parallelism from 32 to 512. Query Q_2^M evaluates just the parallel execution of the `BookOrders` script, while Q_2^{ML} involves a join with the `CUSTOMER` table from the LeanXcale data store:

```
Q2M: SELECT count(*) FROM BookOrders

Q2ML: SELECT count(*)
      FROM BookOrders O, CUSTOMER C
      WHERE O.CUSTKEY = C.C_CUSTKEY
```

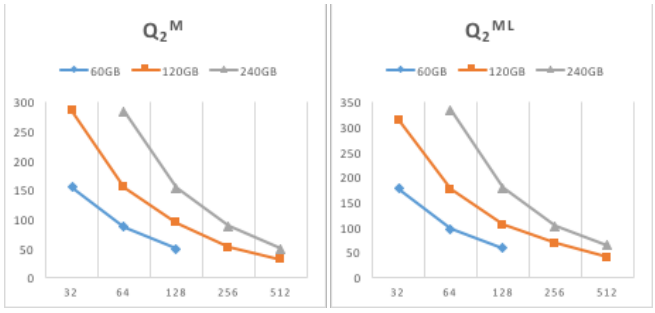


Figure 7. Execution times (in seconds) of Q_2 queries on more sophisticated JavaScript MongoDB subqueries with scales of data from 60 to 240 GB and levels of parallelism from 32 to 512.

Figure 7 shows the performance measurements of Q_2 queries that stress on the evaluation of the parallel processing of highly expressive JavaScript queries, with and without join with a LeanXcale table. Similar conclusions on performance and scalability can be done, like for the Q_1 queries.

C. *Large Scale and Bind Joins*

The third test case evaluates the parallel polyglot query processing in the context of much larger data. Q_3 performs a join between a 600GB version of the `Orders_Items` collection (containing ~770 million documents and ~3 billion order items) and a LeanXcale table `CLICKS` of size 1TB, containing ~6 billion click log records.

```
Q3: SELECT O.CUSTKEY, O.TITLE, C.URL, O.ORDERDATE
    FROM CLICKS C, BookOrders O
    WHERE C.UID = O.CUSTKEY
      AND C.CLICKDATE = O.ORDERDATE
      AND C.IPADDR BETWEEN a AND b
```

The query assumes a use case that aims to find orders of books made on the same day the customers visited the website. The predicate `C.IPADDR BETWEEN a AND b` filters a range of source IP addresses for the web clicks, which results in selecting click data for a particular subset of user IDs. This selectivity makes significant the impact of using bind join within the native table `BookOrders`. The definition of the named table is hence slightly modified, to allow for the bind join to apply early filtering to reduce significantly the amount of data processed by the MongoDB JavaScript subquery:

```
BookOrders(custkey int, orderdate date,
           title string, author string
           JOINED ON custkey
           REFERENCING OUTER AS uids )@mongo =
{*
  return db.orders_items.findSharded(
    {custkey: {$in: uids}} )
  .flatMap( function(doc) {...} );
*}
```

The query executes by first applying the filter and retrieving intermediate data from the `CLICKS` table, which is not indexed, therefore a full scan takes place. The intermediate data are then cached at the workers and a list of distinct values for the `UID` column is pushed to the MongoDB wrapper instances, to form the bind join condition. We use the parameters `a` and `b` to control the selectivity on the large table, hence the selectivity of the bind join. We ran experiments varying the selectivity factor `SF` between 0.02%, 0.2%, and 2%. Smaller values of `SF` result in shorter lists of outer keys for the bind join condition and hence faster execution of the `BookOrders` subquery. However, when not using bind join, the predicate selectivity does not affect significantly the query execution time, as full scans take place on both datasets anyway.

For comparison with Spark SQL, the `CLICKS` dataset was loaded as an HDFS file in order to be accessible by Spark. To run an analogue of the `BookOrders` subquery through the MongoDB connector for Spark SQL, we used the MongoDB aggregation framework against the same sharded collection in our MongoDB cluster as follows:

```
db.orders_items.aggregate([{$unwind: "$items"},
          {$match: {"items.type": "BOOK"}}, ...])
```

Figure 8 shows the times for processing Q3 queries with Spark SQL, with LeanXcale without using bind join, and with LeanXcale using bind join. The level of parallelism for both storing and querying data is 512. Without bind join, Spark SQL shows a slight advantage compared to LeanXcale DQE, which is explainable by the overhead of the JavaScript interpreting that takes place at DQE wrappers for MongoDB. However, the ability for applying bind join that cannot be handled with Spark SQL gives our approach a significant advantage for selective queries, which is very useful in a wide range of industrial scenarios.

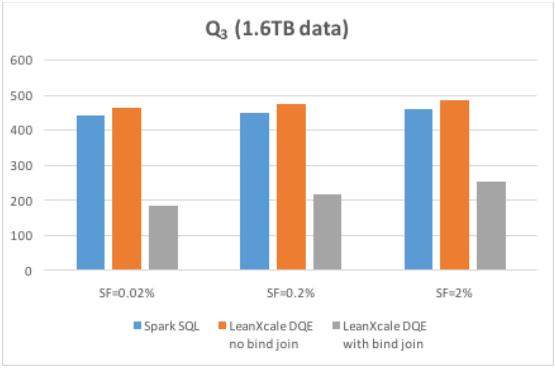


Figure 8. Execution times (in seconds) of Q_3 queries joining an expressive JavaScript MongoDB subquery on a 600GB document collection with a 1TB click logs dataset. The level of parallelism was set to 512, i.e. 512 MongoDB shards, 512 LeanXcale DQE instances, and 512 Spark executors. To assess bind join, SF varied between 0.02%, 0.2%, and 2%.

VI. Comparison with Related Work

The problem of accessing heterogeneous data sources has long been studied in the context of multidatabase and data integration systems [19,23]. More recently, with the advent of cloud databases and big data processing frameworks, multidatabase solutions have evolved towards polystore systems that provide integrated access to a number of RDBMS, NoSQL, NewSQL, and HDFS data stores through a common query engine. Early polystores [17,18,21] typically mediate heterogeneous data stores through a single common data model. The polystore BigDAWG [9,10] 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. Myria [25] is another recent polystore, built on a shared-nothing parallel architecture, that efficiently federates data across diverse data models and query languages. Its extended relational model and the imperative-declarative hybrid language MyriaL span well all the underlying data models, where rewrite rules apply to transform expressions into specific API calls, queries, etc. for each of the data stores. In addition to typical loosely-coupled systems, some polystore solutions [5,7,12] consider the problem of optimal data placement and/or selection of data source, mostly driven by application requirements.

Hybrid polystore systems support data source autonomy as in loosely-coupled systems, and preserve parallelism by exploiting the local data source interface as in tightly-coupled systems. They usually serve as parallel query engines with parallel connectors to external sources. As our work fits in this category, we will briefly discuss some of the existing solutions, focusing on their capabilities to integrate with MongoDB as a representative example of a non-relational data store. However, although they enable parallel integration with data clusters (like MongoDB), none of these systems support the combination of massive parallelism with native queries and the optimizability of bind joins, the way the LeanXcale engine does.

Spark SQL [4] 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. Spark SQL can access a MongoDB cluster through its MongoDB connector that maps a sharded document collection to a DataFrame, partitioned as per the collection's sharding setup. Schema can be either inferred by document samples, or explicitly declared.

Presto [20] is a distributed SQL query engine, running on a cluster of machines, and designed to process interactive analytic queries against data sources of any size. Presto follows the classical MPP (massively parallel processing) DBMS architecture, which, similarly to LeanXcale, consists of a coordinator, multiple workers and connectors (storage plugins that interface external data stores and provide metadata to the coordinator and data to workers). To access a MongoDB cluster, Presto uses a connector that allows the parallel retrieval of sharded collections, which is typically configured with a list of MongoDB servers. Document collections are exposed as tables to Presto, keeping schema mappings in a special MongoDB collection.

Apache Drill [2] is a distributed query engine for large-scale datasets, designed to scale to thousands of nodes and query at low latency petabytes of data from various data sources through storage plugins. The MPP architecture runs a so called "drillbit" service at each node. The drillbit that receives the query from a client or application becomes the foreman for the query and compiles the query into an optimized execution plan, further parallelized in a way that maximizes data locality. The MongoDB storage allows running Drill and MongoDB together in distributed mode, by assigning shards to different drillbits to exploit parallelism. Since MongoDB collections are used directly in the FROM clause as tables, the storage plugin translates relational operators to native MongoDB queries.

Impala [3] is an open-source MPP SQL engine operating over Hadoop data processing environment. As opposed to typical batch processing frameworks for Hadoop, Impala provides low latency and high concurrency for analytical

queries. Impala can access MongoDB collections through a MongoDB connector for Hadoop, designed to provide the ability to read MongoDB data into Hadoop MapReduce jobs.

VII. CONCLUSIONS

In this paper, we introduced a parallel polyglot polystore system that builds on top of LeanXcale's distributed query engine and processes queries in the CloudMdsQL query language. This allows data store native subqueries to be expressed as inline scripts and combined with regular SQL statements in ad-hoc integration statements.

We contribute by adding polyglot capabilities to the distributed data integration engine that takes advantage of the parallel processing capabilities of underlying data stores. We introduced architectural extensions that enable specific native scripts to be handled in parallel at data store shards, so that efficient and scalable parallel joins take place at query engine level. We focused on parallel joins between a relational table and the result of a JavaScript subquery to MongoDB, but the concept relies on an API that allows its generalization to other script engines and data stores as well.

Our experimental validation evaluated the scalability of the query engine by measuring the performance of various join queries. In particular, even in the context of sophisticated JavaScript subqueries, parallel join processing shows good speedup with increase of the parallelism level. This means that, as the volume of data grows, performance can be maintained by simply extending the parallelism to a proportional number of workers and data shards. This evaluation illustrates the benefits of combining the massive parallelism of the underlying data management technologies with the high expressivity of their scripting frameworks and optimizability through the use of bind join, which is the major strength of our work.

ACKNOWLEDGMENT

This research has been partially funded by the European Union's Horizon 2020 Programme, project CloudDBAppliance, grant 732051. The research performed by LeanXcale authors has been also partially funded by the Madrid Regional Council, FSE and FEDER, project Cloud4BigData (S2013TIC2894) and industrial doctorate grant for Pavlos Kranas (IND2017/TIC-7829) and the Ministry of Economy and Competitiveness (MINECO) under projectCloudDB (TIN2016-80350). Prof. Jose Pereira contributed to this work when he was with LeanXcale.

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] Apache Drill – Schema-free SQL Query Engine for Hadoop, NoSQL and Cloud Storage, https://drill.apache.org/

[3] Apache Impala, http://impala.apache.org/

[4] 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*, 1383-1394, 2015.

[5] F. Bugiotti, D. Bursztyn, A. Deutsch, I. Ileana, I. Manolescu. Invisible Glue: Scalable Self-Tuning Multi-Stores. In *Conference on Innovative Data Systems Research (CIDR)*, 2015.

[6] Darema, F. The SPMD model: Past, present and future. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2131. Springer Berlin Heidelberg, 2001.

[7] S. Dasgupta, K. Coakley, A. Gupta. Analytics-driven data ingestion and derivation in the AWESOME polystore. *IEEE International Conference on Big Data*, 2555-2564, 2016.

[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*, 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", *SIGMOD Record,* vol. 44, no. 2, pp. 11-16, 2015.

[10] V. Gadepally, P. Chen, J. Duggan, A. J. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson, M. Stonebraker. The BigDawg polystore system and architecture. In *IEEE High Performance Extreme Computing Conference (HPEC)*, 1-6, 2016.

[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] Y. Khan, A. Zimmermann, A. Jha, D. Rebholz-Schuhmann, R. Sahay. Querying web polystores. *IEEE International Conference on Big Data*, 2017.

[13] B. Kolev, C. Bondiombouy, P. Valduriez, R. Jimenez-Peris, R. Pau, J. Pereira. The CloudMdsQL Multistore System. *ACM SIGMOD*, 2113-2116, 2016.

[14] B. Kolev, R. Pau, O. Levchenko, P. Valduriez, R. Jimenez-Peris, J. Pereira. Benchmarking Polystores: the CloudMdsQL Experience. *IEEE International Conference on Big Data*, 2574-2579, 2016.

[15] 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*, vol. 34, pp. 463-503. Springer, 2015.

[16] J. LeFevre, J. Sankaranarayanan, H. Hacıgümüs, J. Tatemura, N. Polyzotis, M. Carey, "MISO: souping up big data query processing with a multistore system", *ACM SIGMOD*, 1591-1602, 2014.

[17] Z. Minpeng, R. Tore. Querying Combined Cloud-based and Relational Databases. *Int. Conf. on Cloud and Service Computing (CSC)*, 330-335, 2011.

[18] 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.

[19] T. Özsu, P. Valduriez. Principles of Distributed Database Systems. 3rd ed. Springer, 2011, 850 pages.

[20] Presto – Distributed Query Engine for Big Data, https://prestodb.io/

[21] A. Simitsis, K. Wilkinson, M. Castellanos, U. Dayal. Optimizing Analytic Data Flows for Multiple Execution Engines. *ACM SIGMOD*, 829-840, 2012.

[22] M. Stonebraker, U. Cetintemel, One size fits all: An idea whose time has come and gone (abstract). *ICDE*, 2-11, 2005.

[23] 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.

[24] TPC-H. http://www.tpc.org/tpch/

[25] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, D. Moritz, B. Myers, J. Ortiz, D. Suciu, A. Whitaker, S. Xu. The Myria big data management and analytics system and cloud service. In *Conference on Innovative Data Systems Research (CIDR)*, 2017

[26] 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.