Benchmarking Polystores: the CloudMdsQL Experience

Boyan Kolev[1,2], Raquel Pau[3], Oleksandra Levchenko[1], Patrick Valduriez[1], Ricardo Jiménez-Peris[2,4], José Pereira[2,5]
[1] Inria & LIRMM, Montpellier, France
[2] LeanXcale, Madrid, Spain
[3] Sparsity Technologies, Barcelona, Spain
[4] UPM, Madrid, Spain
[5] INESC TEC & U. Minho, Braga, Portugal

***Abstract*—The CloudMdsQL polystore provides integrated access to multiple heterogeneous data stores, such as RDBMS, NoSQL or even HDFS through a big data analytics framework such as MapReduce or Spark. The CloudMdsQL language is a functional SQL-like query language with a flexible nested data model. A major capability is to exploit the full power of each of the underlying data stores by allowing native queries to be expressed as functions and involved in SQL statements. The CloudMdsQL polystore has been validated with a good number of different data stores: HDFS, key-value, document, graph, RDBMS and OLAP engine. In this paper, we introduce the benchmarking of the CloudMdsQL polystore and evaluate the performance benefits of important features enabled by the query language and engine.**

Keywords—benchmark; cloud; heterogeneous data; polystore

I. INTRODUCTION

The blooming of different cloud data management infrastructures, specialized for different kinds of data and tasks, has led to a wide diversification of DBMS interfaces and the loss of a common programming paradigm. This makes it very hard for a user to integrate and analyze her data sitting in different data stores, e.g. RDBMS, NoSQL, and HDFS. The CoherentPaaS project [1] addresses this problem, by providing a rich platform integrating different data management systems specialized for particular tasks, data and workloads. The platform is designed to provide a common programming model and language to query multiple data stores.

The problem of accessing heterogeneous data sources has long been studied in the context of multidatabase and data integration systems [7]. More recently, with the advent of cloud databases and big data processing frameworks, the solution has evolved towards polystores (also called multistore systems) that provide integrated access to a number of RDBMS, NoSQL and HDFS data stores through a common query engine. Data mediation SQL engines, such as Apache Drill, Spark SQL, and SQL++ provide common interfaces that allow different data sources to be plugged in (through the use of wrappers) and queried using SQL. The BigDAWG polystore [3] goes farther by enabling queries across "islands of information", where each island corresponds to a specific data model and its language and provides transparent access to a subset of the underlying data stores through the island's data model. Another family of multistore systems [2,6] has been introduced with the goal of tightly integrating big data analytics frameworks (e.g. MapReduce or Spark) with traditional RDBMS, by sacrificing the extensibility with other data sources. However, since none of these approaches supports the ad-hoc usage of native queries, they do not preserve the full expressivity of an arbitrary data store's query language. But what we want to give the user is the ability to express powerful ad-hoc queries that exploit the full power of the different data store languages, e.g. directly express a path traversal in a graph database.

The Cloud Multidatastore Query Language (CloudMdsQL) is a functional SQL-like query language, designed to serve the querying capabilities of the CoherentPaaS platform. A CloudMdsQL query can address multiple heterogeneous databases by means of nested subqueries [5]. Each subquery addresses directly a particular data store and may contain embedded invocations to the data store's native query interface. Thus, the major innovation is that a CloudMdsQL query can exploit the full power of local data stores, by simply allowing some local data store native queries (e.g. a breadth-first search query against a graph database) to be called as functions, and at the same time be optimized, e.g. by pushing down select predicates, using bind join, etc.

One of the major challenges in front of the CloudMdsQL language/engine is to allow joins across heterogeneous data stores and to be able to perform them in an efficient way. For this reason, we pay special attention to the use of bind joins [4] and we apply this technique even when native queries are used. In this paper, we introduce the benchmarking of the CloudMdsQL system, which evaluates the ability to run optimized queries across heterogeneous data stores, as well as the functional and performance benefits of the usage of native queries in combination with SQL statements.

The rest of the paper is organized as follows. Section 2 gives an overview of the CloudMdsQL language and engine. Section 3 presents the benchmark environment and the results of the performance evaluation. Section 4 concludes.

This research has been partially funded by the European Commission under projects CoherentPaaS, LeanBigData, and CloudDBAppliance (grants FP7-611068, FP7-619606, and H2020-732051), the Madrid Regional Council, FSE and FEDER, project Cloud4BigData (grant S2013TIC-2894), the Spanish Research Agency MICIN project BigDataPaaS (grant TIN2013-46883), and the Spanish CDTI/MEC NEOTEC program (grant SNEO-20151285).

II. CLOUDMDSQL OVERVIEW

The CloudMdsQL language is SQL-based with the extended capabilities for embedding subqueries expressed in terms of each data store's native query interface. The common data model respectively is table-based, with support of rich datatypes that can capture a wide range of the underlying data stores' datatypes, such as arrays and JSON objects, in order to handle non-flat and nested data, with basic operators over such composite datatypes.

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

A. Query Language

Queries that integrate data from several data stores usually consist of subqueries and an integration SELECT statement. A subquery is defined as a named table expression, i.e. an expression that returns a table and has a name and signature. The signature defines the names and types of the columns of the returned relation. Thus, each query, although agnostic to the underlying data stores' schemas, is executed in the context of an ad-hoc schema, formed by all named table expressions within the query. A named table expression can be defined by means of either an SQL SELECT statement (that the query compiler is able to analyze and possibly rewrite) or a native expression (that the query engine considers as a black box and delegates its processing directly to the data store). For example, the following simple CloudMdsQL query contains two subqueries, defined by the named table expressions `T1` and `T2`, and addressed respectively against the data stores `rdb` (an SQL database) and `mongo` (a MongoDB database):

```
T1(x int, y int)@rdb = ( SELECT x, y FROM A )
T2(x int, z array)@mongo = {*
  db.B.find( {$lt: {x, 10}}, {x:1, z:1, _id:0} )
*}
SELECT T1.x, T2.z
FROM T1, T2
   WHERE T1.x = T2.x AND T1.y <= 3
```

The purpose of this query is to perform relational algebra operations (expressed in the main SELECT statement) on two datasets retrieved from a relational and a document database. The two subqueries are sent independently for execution against their data stores in order the retrieved relations to be joined by the CloudMdsQL query engine. The SQL table expression `T1` is defined by an SQL subquery, while `T2` is a native expression (identified by the special bracket symbols `{* *}`) expressed as a native MongoDB call. Note that subqueries to some NoSQL data stores can also be expressed as SQL statements; in such cases, the wrapper must provide the mapping from relational operators to native calls. Within our evaluation, unlike in the example above, we use an SQL wrapper to query MongoDB, which also benefits from subquery rewriting.

CloudMdsQL allows named table expressions to be defined as Python functions, which is useful for querying data stores that have only API-based query interface. A Python expression yields tuples to its result set much like a user-defined table function. It can also use as input the result of other subqueries. Furthermore, named table expressions can be parameterized by declaring parameters in the expression's signature. For example, the following Python expression uses the intermediate data retrieved by `T2` to return another table containing the number of occurrences of the parameter `v` in the array `T2.z`.

```
T3(x int, c int WITHPARAMS v string)@python = {*
  for (x, z) in CloudMdsQL.T2:
    yield( x, z.count(v) )
*}
```

A (parameterized) named table can then be instantiated by passing actual parameter values from another native/Python expression, as a table function in a FROM clause, or even as a scalar function (e.g. in the SELECT list). Calling a named table as a scalar function is useful e.g. to express direct lookups into a key-value data store. In fact, in the current benchmark, we evaluate the usage of scalar lookups as an efficient alternative to the usage of expensive joins.

Note that parametrization and nesting is also available in SQL and native named tables. Within our evaluation, we give an example that involves the Sparksee graph database and we use its Python API to express subqueries that benefit from all of the features described above. In fact, our initial query engine implementation enables Python integration; however support for other languages (e.g. JavaScript) for user-defined operations can be easily added.

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 … 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)
```

In the above example, using bind join will be reasonable only in the presence of an index on the column `a.id` in data

store DB1, because such a presence will make the pushed down IN operator avoid an expensive table scan.

The way to do the bind join analogue for native/Python queries is through the use of a JOINED ON clause in the named table signature. For example, if A is defined as the Python function below, as A.id participates in an equi-join, the values b_1 … b_n will be provided to the Python code through the iterator b_keys (in this context, we refer to the table B as the "outer" table, and b_keys as the outer keys):

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

In fact, each wrapper may provide different mechanisms for the programmer to consume the values of the outer keys. In this example, as the native query is expressed in an imperative language, the set of outer keys is represented by an iterator object; however, for declarative native queries, more appropriate will be to define a placeholder, which the wrapper will substitute with the textual representation of the set of values.

C. Query Engine

For the current implementation of the query engine, we modified the open source Apache Derby database to accept CloudMdsQL queries and transform the corresponding execution plan into Derby SQL operations. We developed the query planner and the query execution controller and linked them to the Derby core, which we use as the operator engine. Derby allows extending the set of SQL operations by means of CREATE FUNCTION statements. This type of statements creates an alias, with an optional set of parameters, to invoke a specific Java component as part of an execution plan. Thus, for each named table expression in a query, a table function is created dynamically, which invokes the corresponding wrapper as a Java class. Thus, Derby handles global execution, delegating local optimization and execution to the underlying data stores. As a second step, the query engine evaluates which named expressions are queried more than once and must be cached into the temporary table storage, which will be always queried and updated from the specified Java functions to reduce the query execution time. Finally, the last step consists of translating all operation nodes that appear in the execution plan into a Derby specific SQL execution plan.

III. Benchmarking and Performance Evaluation

The goal is to assess the ability of the query engine to perform optimized CloudMdsQL queries across data stores. To evaluate the performance improvements, we compare the execution times (with and without optimization) of queries that involve a quite diverse set of data stores. We divide the performed test cases in two groups that focus on the performance benefits thanks to: (1) bind joins and (2) the support of native queries.

A. Data Stores

We performed the experiments on 5 different data stores, each having different interfaces and data models, in order to have a representative data store in each of the categories: relational, key-value, document, graph databases, and an OLAP engine. They are summarized in this subsection, detailing the aspects that are important for the evaluation, namely: (1) the data model, (2) the query language, (3) whether SQL or native queries are used for subquerying, and (4) how the wrapper handles the set of outer keys to provide bind join support.

MonetDB is a column-store database with an SQL engine atop. It follows a relational data model and is queried by SQL through CloudMdsQL subqueries. To handle the outer keys for bind join, the wrapper creates a temporary table (say outer_keys) in the MonetDB database and populates it with the values of the outer keys; then it rewrites the subquery by adding the predicate IN (SELECT * FROM outer_keys). If the bind join is on multiple keys, the generated predicate is different and uses the EXISTS operator.

MongoDB is a document database that has a Java based query interface, where the chain of operations is specified through the use of API calls with JSON documents as parameters that typically return document collections. Since in some particular but very common cases (e.g. collections of flat documents with fixed field names), the document data model can be considered as a subset of the relational model, it is possible to map simple SQL commands to native MongoDB queries. For this reason, our MongoDB wrapper provides a translation from relational operations to native API calls, so that MongoDB can be subqueried by SQL statements. In addition, it also supports native MongoDB queries. When consuming the outer keys for bind join, the wrapper dynamically builds a filter condition using the MongoDB $in operator and applies it through a call to the find() function.

Apache HBase is a key-value data store with a very simple interface mostly consisting of primitives to *scan* a whole table or to *get* the value for a particular key. Within CloudMdsQL, subqueries to HBase are expressed with native statements that can be of the form of either (1) scan '<table_name>' or (2) get '<table_name>', <key>. Therefore, there is no way to directly apply an equivalent of an IN operator to HBase subqueries, but the optimized functionality of a bind join can be achieved through the use of scalar lookups.

Sparksee is a graph database that has a Python API and a declarative notation for graph algebra expressions. Within CloudMdsQL, subqueries to Sparksee are expressed in Python code. When handling the outer keys for bind join, the wrapper provides to the Python code an iterator object that the programmer can use to get the values of the outer keys and use them in native API calls to apply the filter.

ActivePivot is an OLAP engine where data are organized in cubes and queried through MDX statements. Subqueries within CloudMdsQL are expressed as native MDX commands. To handle the outer keys for bind join, the programmer uses a special placeholder in the MDX expression, which the wrapper identifies and dynamically replaces with the textual representation of the set of values of the outer keys.

B. Experimental Setup

We performed our experiments using data based on the TPC-H benchmark schema (www.tpc.org/tpch). Each data store contains the same 8 datasets, generated by the TPC-H generator, stored in tables or in the corresponding for the data store structures, as follows. In MonetDB the TPC-H datasets are stored as relational tables, in MongoDB – as document collections. In HBase they are organized in HBase tables, where each key-value pair corresponds to a row from the dataset; in particular, the key corresponds to the value(s) of the key column(s), while the value contains the serialization of the entire row. In Sparksee, each of the TPC-H datasets is loaded into a set of graph nodes of the same type, while relationships between them are mapped to graph edges, i.e., a join between two datasets is expressed as a graph neighborship query. In ActivePivot, all the generated TPC-H data are loaded in one single cube.

For the experiments, we use a cluster of 6 identical machines (8GB RAM, 4 CPU cores @2.4GHz), of which one node is dedicated to the CloudMdsQL engine and one node per each of the data stores. Each data store node contains data generated as per the TPC-H schema, with scale factor 1. Our queries use the datasets `LINEITEM` and `ORDERS`, where the columns `L_ORDERKEY`, `L_PARTKEY`, `O_ORDERKEY`, `O_ORDERSTATUS` and `O_ORDERPRIORITY` are indexed.

Our queries use the datasets `LINEITEM` and `ORDERS`, where the columns `L_ORDERKEY`, `L_PARTKEY`, `O_ORDERKEY`, `O_ORDERSTATUS` and `O_ORDERPRIORITY` are indexed.

C. Evaluation of Bind Joins

With this set of test cases, we focus on the evaluation of performance benefits thanks to the use of bind joins, which is an efficient technique to apply semi-joins between datasets from different data stores by allowing the output of one of the subqueries to be used by the other subquery to filter out the unnecessary for the join rows. As we apply this technique to a quite diverse set of data stores, even if they do not natively support SQL querying, we consider this evaluation very important for the benchmarking of the CloudMdsQL system.

We consider the following query, assuming that the column `O_ORDERKEY` is indexed. The selection predicate `L_PARTKEY = 10` results in retrieving only 24 rows from the `LINEITEM` table which are then supposed to be joined with the `ORDERS` table, which contains 1.5 million rows. The non-optimized execution of this query implies that the entire `ORDERS` table is retrieved at the common query engine before being joined with the small table. In the optimized variant, the `T2` subquery is rewritten by adding the predicate `O_ORDERKEY IN (`k_1`, `k_2`, …, `k_{24}`)`, where k_i are the values of the column `L_ORDERKEY`, taken from the small table `T1`. Thus, only the rows that match the join condition (24 instead of 1.5 million) will be retrieved from the `ORDERS` table.

```
T1( L_ORDERKEY long, L_PARTKEY long, L_SUPPKEY long,
  L_LINENUMBER int, L_QUANTITY float,
  L_EXTENDEDPRICE float, L_DISCOUNT float,
  L_TAX float, L_RETURNFLAG string,
  L_LINESTATUS string, L_SHIPDATE date,
  L_COMMITDATE date, L_RECEIPTDATE date,
  L_SHIPINSTRUCT string, L_SHIPMODE string,
  L_COMMENT string)@datastore1 =
(
  SELECT L_ORDERKEY, L_PARTKEY, L_SUPPKEY,
    L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE,
    L_DISCOUNT, L_TAX, L_RETURNFLAG, L_LINESTATUS,
    L_SHIPDATE, L_COMMITDATE, L_RECEIPTDATE,
    L_SHIPINSTRUCT, L_SHIPMODE, L_COMMENT
  FROM LINEITEM
  WHERE L_PARTKEY = 10
)

T2( O_ORDERKEY long, O_CUSTKEY long,
    O_ORDERSTATUS string, O_TOTALPRICE float,
    O_ORDERDATE date, O_ORDERPRIORITY string,
    O_CLERK string, O_SHIPPRIORITY int,
    O_COMMENT string)@datastore2 =
(
    SELECT O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
      O_TOTALPRICE, O_ORDERDATE, O_ORDERPRIORITY,
      O_CLERK, O_SHIPPRIORITY, O_COMMENT
    FROM ORDERS
)

SELECT T1.*,
    T2.O_CUSTKEY, T2.O_ORDERSTATUS, T2.O_TOTALPRICE,
    T2.O_ORDERDATE, T2.O_ORDERPRIORITY, T2.O_CLERK,
    T2.O_SHIPPRIORITY, T2.O_COMMENT
FROM T1 JOIN T2 ON L_ORDERKEY = O_ORDERKEY
```

The above query, as specified, is applicable only if both `datastore1` and `datastore2` support SQL subquerying (MonetDB and MongoDB). To apply the bind join method, the query compiler takes care of doing subquery rewriting by pushing down to the subquery for `T2` a predicate condition containing an `IN` operator that takes the values from an intermediate named table, containing the set of outer keys. As described in the previous subsection, the wrappers for MonetDB and MongoDB use different approaches to achieve the goal.

Next, we will explore the performance of this approach when using native subqueries, especially when `T2` is native, as it requires the native query to access intermediate data from the table storage of the common query engine. This is done with the help of the programmer by adding to the signature of `T2` the following clause:

```
JOINED ON O_ORDERKEY REFERENCING OUTER AS t1_keys
```

Thus, whenever `T2` is used for a bind join, the join key values of the other side of the join (the outer keys, taken from `T1`) are provided in the intermediate table `t1_keys` and the native query for `T2` needs to use the corresponding mechanism that its wrapper provides to access these join keys and use them to return only rows that match the join criteria. Sparksee and ActivePivot wrappers provide different mechanisms for expressing this, as illustrated in the corresponding implementations of `T2` below (for simplicity, the full signature of `T2` is not repeated and some notations are shortened):

```
T2( ... JOINED ON O_ORDERKEY
    REFERENCING OUTER AS t1_keys)@sparksee =
{*
  rs = CloudMdsQL.t1_keys()
  while rs.next():
    rs2 = graph.compute( \
      "GRAPH::SELECT( 'ORDERS'.'O_ORDERKEY' = " + \
        rs[1] + ")")
```

```
    while rs2.next():
      yield( rs2[1], rs2[2], ..., rs2[9] )
    rs2.close();
  rs.close()
*}
```

The above Python code for Sparksee iterates through the outer keys and for each key finds the corresponding graph node and yields a tuple with its attributes, which correspond to the columns of the ORDERS dataset.

```
T2( ... JOINED ON O_ORDERKEY
    REFERENCING OUTER AS t1_keys)@activepivot =
{*
  SELECT NON EMPTY CROSSJOIN(
    [Orders].[O_ORDERKEY].Members,
    [Orders].[O_CUSTKEY].Members,
    ..........,
    [Orders].[O_COMMENT].Members)
  ON ROWS FROM
    (SELECT {$$£[Orders].[O_ORDERKEY].£t1_keys§§}
  ON COLUMNS FROM [TPCHCube])
*}
```

The above MDX query contains the special placeholder `$§£[Orders].[O_ORDERKEY].£t1_keys§§`, which the ActivePivot wrapper uses as a template to dynamically generate the list of the outer keys and hence substitute the placeholder with a list of values like: `[Orders].[O_ORDERKEY].[36341], ...`

The above approaches allow for the native query to filter out rows by applying an analogue of the `IN` operator. This is possible, because Sparksee and ActivePivot provide powerful query mechanisms. However, for simpler query notations, such as the one for the key-value data store HBase, this approach is not applicable, as HBase has only primitives either to retrieve a whole table (*scan*) or to make a lookup for the value for a particular key (*get*) or ranges of keys. That is why, to achieve the same benefits of bind join as above, when `T2` is an HBase query, we take advantage of scalar lookups, that allow a parameterized named table (`T2`) to be used as a scalar function and evaluated for every value of a column from another table (`T1`). So, `T2` is defined as a parameterized function that gets all the values of `ORDERS` for a particular key. Then, `T2` is called in the `SELECT` list of the main `SELECT` statement of the query, instead of being joined with `T1`. The implementation of `T2` for HBase and the main `SELECT` statement for this case are as follows:

```
T2( ... WITHPARAMS orderkey long)@hbase = {*
    get 'orders', orderkey
*}

SELECT T1.*, T2(L_ORDERKEY).O_CUSTKEY,
  T2(L_ORDERKEY).O_ORDERSTATUS,
  ..........,
  T2(L_ORDERKEY).O_COMMENT
FROM T1
```

We performed our experiments in the context of the above query and its variations with respect to the `T2` subquery for the 5 different data stores. Since the retrieval of the `T1` table is quite trivial and its retrieval time is similar for all data stores, we do not focus on various data stores for the `T1` subquery and performed all the experiments with MongoDB as `datastore1`, varying `datastore2` among all the 5 data stores.

We also experimented with 5 different selectivity factors of the bind join condition by changing the `T1` filter predicate (initially `L_PARTKEY = 10`), thus varying the number of rows returned by `T1`, which results in different cardinalities of the set of outer keys. The different filter predicates we used and their corresponding selectivity factors (SF) are detailed in Table 1.

TABLE I. BIND JOIN SELECTIVITIES

Filter predicate	Rows (approx.)	SF Rows / 1.5M
`L_PARTKEY = 10`	24	0.002%
`L_PARTKEY <= 10`	300	0.02%
`L_PARTKEY <= 100`	3000	0.2%
`L_PARTKEY <= 1000`	30000	2%
`L_PARTKEY <= 10000`	300000	20%

Thus, the results of the current or similar benchmark can serve as a statistical base to determine the profitability of bind join compared to traditional join for each data store as a function of the join selectivity, as well as to determine a selectivity threshold, below which using bind join will be considered reasonable. Since each data store and wrapper uses different mechanisms to handle the outer keys, these metrics are quite different for each of the 5 data stores.

Table 2 summarizes the results of the performed evaluation on the bind join test cases with the 5 data stores and the 5 different selectivity factors. For each test case, the summary includes the execution times (in milliseconds) for performing traditional join (TJ) compared to bind join (BJ) and the performance improvement (PI = TJ/BJ). The test cases, for which there is no performance improvement, are omitted.

TABLE II. BIND JOIN PERFORMANCE BENEFITS

Data Store		Selectivity factor				
		0.002%	***0.02%***	***0.2%***	***2%***	***20%***
MonetDB	TJ	56582	58113	59294	62027	67509
	BJ	953	1120	1982	8938	68757
	PI	**59.4**	**51.9**	**29.9**	**6.9**	**1.0**
MongoDB	TJ	63783	65654	67807	75218	81146
	BJ	233	363	1124	6126	40908
	PI	**273.7**	**180.9**	**60.3**	**12.3**	**2.0**
HBase	TJ	100576	102218	103340	105123	
	BJ	1520	2711	10582	75886	
	PI	**66.2**	**37.7**	**9.8**	**1.4**	
Sparksee	TJ	149694	151239	152116		
	BJ	1480	7409	57739		
	PI	**101.1**	**20.4**	**2.6**		
ActivePivot	TJ	205071	206803	207944		
	BJ	1739	14409	145788		
	PI	**117.9**	**14.4**	**1.4**		

Bind join produces an overhead, which is why for high values of the selectivity factor its usage is not advisable. First, when using bind join, the query engine cannot retrieve both tables in parallel, as it must wait for the outer table to be fully retrieved in order to get the set of distinct values of the outer keys. Second, it requires that this set of values be pushed into the subquery, which is handled differently by each data store, which explains the variability of performance improvements. The MonetDB wrapper makes a modification of the schema to create and populate a temporary table. MongoDB is doing

faster, as it simply creates a filter out of the outer keys. With HBase, the process is much slower, as a native query is initiated by the query engine for each row of the outer table. The case of Sparksee is similar, as the Python code requests that a graph query be evaluated for each of the outer keys. And for ActivePivot, query rewriting takes place, however it brings a significant textual overhead, as the values of the outer keys must be specified with full dimension path.

D. Evaluation of Native Query Support

The support of native queries in CloudMdsQL allows the user to write powerful subqueries to the data stores that efficiently perform specific for the data store operations with native statements, rather than to express them in SQL and handle at the common query engine. This requires deeper expertise and knowledge about the specifics of the queried data stores; however, the added value is that the programmer can still request efficient operations at the data store, thanks to the support of native queries, and combine the results with other data store queries.

Let us consider the following queries, which request different operations to a data store, expressed in SQL. The queries focus on selection, aggregation, grouping, sorting and join operations.

```
Q_SEL:  SELECT L_ORDERKEY FROM LINEITEM
        WHERE L_PARTKEY = 10

Q_AGG:  SELECT COUNT(*) FROM LINEITEM
        WHERE L_QUANTITY < 5

Q_GRP:  SELECT O_ORDERSTATUS, O_ORDERPRIORITY,
          AVG (O_TOTALPRICE)
        FROM ORDERS
        GROUP BY O_ORDERSTATUS, O_ORDERPRIORITY

Q_ORD:  SELECT O_ORDERSTATUS, O_ORDERPRIORITY,
          AVG (O_TOTALPRICE)
        FROM ORDERS
        GROUP BY O_ORDERSTATUS, O_ORDERPRIORITY
        ORDER BY O_ORDERSTATUS, O_ORDERPRIORITY

Q_JOIN: SELECT O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS,
          L_QUANTITY
        FROM LINEITEM JOIN ORDERS
          ON L_ORDERKEY = O_ORDERKEY
        WHERE L_PARTKEY = 10
```

Let us assume that the programmer needs to request the same operations to non-SQL data stores (Sparksee and ActivePivot) that can natively support them. The naïve way to achieve this is for the wrapper developer or DBA to provide a static mapping of data store structures to relational tables and let the programmer express her query in SQL at the common query engine level. But now the programmer can take advantage of the native queries support and express the operations in native for the data store statements, which will result in an optimal execution. Note that all the generated TPC-H data for ActivePivot are stored in one single cube; so, although ActivePivot does not natively support joins between cubes, we categorize it as a join-capable data store only for this test case, because in fact a conceptual join between `LINEITEM` and `ORDERS` is handled by ActivePivot by dealing with dimensions of the same cube.

TABLE III. NATIVE QUERY SUPPORT PERFORMANCE BENEFITS

Data Store		*Q_SEL*	*Q_AGG*	*Q_GRP*	*Q_ORD*	*Q_JOIN*
Sparksee	SQ	19068	26893	36870	51075	69393
	NQ	76	326	2654	2640	785
	PI	**250.9**	**82.5**	**13.9**	**19.3**	**88.4**
Active Pivot	SQ	20009	12592	10067	10907	193963
	NQ	261	112	16	77	189
	PI	**76.7**	**112.4**	**629.2**	**141.6**	**1026.3**

With this group of test cases, we compare the execution times of the queries in two variants: when the requested operations are expressed in native queries and done by the data store (NQ), and when the operations are expressed in SQL queries and executed by the query engine after the wrapper simply delivers the entire table (SQ). Table 3 shows the comparison between NQ and SQ execution times, as well as the performance improvement (PI = SQ/NQ).

IV. CONCLUSION

This paper introduced the benchmarking of the CloudMdsQL polystore and evaluated the performance benefits in two specific categories of queries. First, the evaluation of bind join shows the ability of the query engine to run efficient queries across heterogeneous databases using both SQL and native approaches. Moreover, the measurements of the current or similar benchmark can be used as provision for a cost model based on inferring the performance improvement of bind joins for each particular data store. Second, the evaluation of the native query support stresses on the performance benefits provided by the high expressivity of the query language. With this benchmark we focused mostly on relational operators and their equivalents in non-SQL data stores. As a further work we will concentrate on evaluating the performance benefits in the context of data store specific operators, such as graph traversals or queries on nested documents.

REFERENCES

[1] CoherentPaaS, http://coherentpaas.eu

[2] D. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, J. Gramling, "Split query processing in Polybase", ACM SIGMOD. 2013, pp. 1255-1266.

[3] 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 Rec. 44, 2. 2015, pp. 11-16.

[4] L.M. Haas, D. Kossmann, E.L. Wimmers, and J. Yang, "Optimizing queries across diverse data sources", Int. Conf. on Very Large Databases (VLDB). 1997, pp. 276-285.

[5] 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. Springer 2015, pp. 463-503.

[6] 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. 2014, pp. 1591-1602.

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